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

344425-004US

June 2022

[Category]

Notices and Disclaimers

Table of Contents

SECTION 1: INTRODUCTION AND OVERVIEW 12

1. About this Document 13

1.1. Scope of this Document 13

1.2. Document Organization 13

1.3. Glossary 13

1.4. Notation 16

1.4.1. Requirement and Definition Commitment Levels 16

1.5. References 17

1.5.1. Intel Public Documents 17

1.5.2. Intel TDX Public Documents 17

2. Overview of Intel® Trust Domain Extensions 18

2.1. Intel TDX Module Lifecycle 18

2.1.1. Boot-Time Configuration and Intel TDX Module Loading 18

2.1.2. Intel TDX Module Initialization, Enumeration and Configuration 18

2.2. Guest TD Life Cycle Overview 19

2.2.1. Guest TD Build 19

2.2.2. Guest TD Execution 19

2.2.3. Guest TD Management during its Run-Time 19

2.3. Intel TDX Operation Modes and Transitions 20

2.4. Guest TD Private Memory Protection 21

2.4.1. Memory Encryption 21

2.4.2. Address Translation 21

2.5. Guest TD State Protection 22

2.6. Intel TDX I/O Model 22

2.7. Measurement and Attestation 23

2.8. Intel TDX Managed Control Structures 23

2.9. Intel TDX Interface Functions 24

2.9.1. Host-Side (SEAMCALL Leaf) Interface Functions 24

2.9.2. Guest-Side (TDCALL Leaf) Interface Functions 26

3. Software Use Cases 27

3.1. Intel TDX Module Lifecycle 27

3.1.1. Intel TDX Module Platform-Scope Initialization 27

3.1.2. Intel TDX Module Shutdown and Update 27

3.2. TD Build 28

3.3. TD Run Time 30

3.3.1. Private Memory Management 30

3.3.1.1. Dynamic Page Addition (Shared to Private Conversion) 30

3.3.1.2. Dynamic Page Removal (Private to Shared Conversion) 31

3.3.1.3. Page Promotion (Mapping Merge) 32

3.3.1.4. Page Demotion (Mapping Split) 33

3.3.1.5. GPA Range Unblock 34

3.3.2. Guest TD Execution 34

3.3.2.1. TD VCPU First-Time Invocation 34

3.3.2.2. TD VCPU Entry, Exit on TDG.VP.VMCALL and Re-Entry 35

3.3.2.3. TD VCPU Entry, Exit on Asynchronous Event and Re-Entry 35

3.3.2.4. Guest-Side Functions 36

3.3.2.5. TD VCPU Rescheduling (Migration to Another LP) ... 37
3.4. TD Destruction ... 38

SECTION 2: INTEL TDX MODULE ARCHITECTURE SPECIFICATION ... 40

4. Key Management ... 41
4.1. Objectives ... 41
4.2. Background: HKID Space Partitioning ... 41
4.3. Key Management Tables ... 42
4.4. Combined Key Management State ... 43
4.5. Key Management Sequences ... 44
4.5.1. Intel TDX Module Initialization: Setting an Ephemeral Key and Reserving an HKID for Intel TDX Data ... 44
4.5.2. TD Creation, Keys Assignment and Configuration ... 44
4.5.3. TD Keys Reclamation, TLB and Cache Flush ... 45

5. TD Non-Memory State (Metadata) and Control Structures ... 46
5.1. Overview ... 46
5.1.1. Opaque vs. Shared Control Structures ... 46
5.1.2. Scope of Control Structures ... 46
5.2. TD-Scope Control Structures ... 46
5.2.1. TDR (Trust Domain Root) ... 47
5.2.2. TDCS (Trust Domain Control Structure) ... 47
5.3. TD VCPU-Scope Control Structures and Management Functions ... 47
5.3.1. Trust Domain Virtual Processor State (TDVPS) ... 47
5.3.1.1. Physical View of TDVPS: TDVPR/TDVPX ... 48
5.3.1.2. Logical View of TDVPS ... 48
5.3.2. Non-Protected Control Structures: Shared EPT and VMCS Auxiliary Control Structures ... 49
5.4. TD Non-Memory State (Metadata) Access Functions ... 49
5.5. Concurrency Restrictions and Enforcement ... 49

6. TD Life Cycle Management ... 50
6.1. TD Life Cycle State Machine ... 50
6.2. TD Creation Sequence ... 50
6.3. VCPU Creation and Initialization Sequence ... 51
6.4. TD Teardown Sequence ... 52

7. Physical Memory Management ... 53
7.1. Trust Domain Memory Regions (TDMRs) and Physical Address Metadata Tables (PAMTs) ... 53
7.2. TDMR Details ... 53
7.3. PAMT Details ... 53
7.3.1. PAMT Entry ... 53
7.3.2. PAMT Blocks and PAMT Arrays ... 54
7.3.3. PAMT Hierarchy and Page Types ... 55
7.4. Adding Physical Pages ... 56
7.4.1. Preventing Cache Line Aliasing ... 56
7.4.2. Adding Pages not Mapped to the Guest TD ... 56
7.4.3. Adding Pages and Mapping to the Guest TD's GPA ... 56
7.5. Reclaiming Physical Pages ... 56
7.5.1. Reclaiming Pages not Mapped to the Guest TD ... 56
7.5.2. Reclaiming TD Pages in TD_TEARDOWN State ... 56

7.5.3. Reclaiming Physical Pages as Part of TD Private Memory Management 57

8. TD Private Memory Management 58

8.1. Overview 58

8.2. Secure EPT Entry 59
8.2.1. Overview 59
8.2.2. SEPT Entry State Diagrams 59

8.3. EPT Walk 60

8.4. Secure EPT Induced TD Exits 61

8.5. Secure EPT Induced Exceptions 61

8.6. Secure EPT Concurrency 61

8.7. Introduction to TLB Tracking 62

8.8. Secure EPT Build and Update: TDH.MEM.SEPT.ADD 62

8.9. Adding TD Private Pages during TD Build Time: TDH.MEM.PAGE.ADD 64

8.10. Dynamically Adding TD Private Pages 64
8.10.1. Overview 64
8.10.2. Page Addition by the Host VMM: TDH.MEM.PAGE.AUG 65
8.10.3. Page Acceptance by the Guest TD: TDG.MEM.PAGE.ACCEPT 66
8.10.3.1. Description 66
8.10.3.2. TDG.MEM.PAGE.ACCEPT Concurrency 67

8.11. Page Merge: TDH.MEM.PAGE.PROMOTE 68

8.12. Page Split: TDH.MEM.PAGE.DEMOTE 69

8.13. Relocating TD Private Pages: TDH.MEM.PAGE.RELOCATE 70

8.14. Removing TD Private Pages: TDH.MEM.PAGE.REMOVE 71

8.15. Removing a Secure EPT Page: TDH.MEM.SEPT.REMOVE 71

8.16. Unblocking a GPA Range: TDH.MEM.RANGE.UNBLOCK 72

9. TD VCPU 73

9.1. VCPU Transitions 73
9.1.1. Initial TD Entry, Asynchronous TD Exit and Subsequent TD Entry 73
9.1.2. Synchronous TD Exit and Subsequent TD Entry 74
9.1.3. VCPU Activity State Machine 74

9.2. TD VCPU TLB Address Space Identifier (ASID) 75
9.2.1. TD ASID Components 75
9.2.2. INVEPT by the Host VMM for Managing the Shared EPT 76

9.3. VCPU-to-LP Association 76
9.3.1. Non-Coherent Caching 76
9.3.2. Intel TDX Functions for VCPU-LP Association and Dis-Association 76
9.3.3. Performance Considerations 77

10. CPU Virtualization (Non-Root Mode Operation) 78

10.1. Initial State 78
10.1.1. Overview 78
10.1.2. Initial State of Guest TD GPRs 78
10.1.3. Initial State of CRs 78
10.1.4. Initial State of Segment Registers 79
10.1.5. Initial State of MSRs 79

10.2. Guest TD Run Time Environment Enumeration 79

10.3. CPU Mode Restrictions 80

10.4. Instructions Restrictions ... *80*
10.4.1. Instructions that Cause a #UD Unconditionally ... 80
10.4.2. Instructions that Cause a #VE Unconditionally ... 80
10.4.3. Instructions that Cause a #UD or #VE Depending on Feature Enabling ... 80
10.4.4. Other Cases ... 81

10.5. Extended Feature Set ... *81*
10.5.1. Allowed Extended Features Control ... 81
10.5.2. Extended State Isolation ... 81
10.5.3. Extended Features Execution Control ... 81

10.6. CR Handling ... *83*
10.6.1. CR0 ... 83
10.6.2. CR4 ... 84

10.7. MSR Handling ... *84*
10.7.1. Overview ... 84

10.8. CPUID Virtualization ... *84*
10.8.1. CPUID Configuration by the Host VMM ... 84
10.8.2. Unconditional #VE for all CPUID Leaves and Sub-Leaves ... 85

10.9. Interrupt Handling and APIC Virtualization ... *85*
10.9.1. Virtual APIC Mode ... 85
10.9.2. Virtual APIC Access by Guest TD ... 85
10.9.3. Implicit APIC Write #VE ... 87
10.9.4. Posted Interrupts ... 87
10.9.5. Pending Virtual Interrupt Delivery Indication ... 87
10.9.6. Cross-TD-VCPU IPI ... 88
10.9.7. Virtual NMI Injection ... 88

10.10. Virtualization Exception (#VE) ... *88*
10.10.1. Virtualization Exception Information ... 88
10.10.2. #VE Injection by the CPU due to EPT Violations ... 89
10.10.3. #VE Injected by the Intel TDX Module ... 89

10.11. Secure and Shared Extended Page Tables (EPTs) ... *89*
10.11.1. GPAW-Relate EPT Violations ... 90
10.11.2. EPT Violation Mutated into #VE ... 90

10.12. Prevention of TD-Induced Denial of Service ... *90*
10.12.1. Bus Lock Detection by the TD OS ... 90
10.12.2. Impact of MSR_TEST_CTRL (MSR 0x33) ... 90
10.12.3. Bus Lock TD Exit ... 90
10.12.4. Notification TD Exit ... 91

10.13. Time Stamp Counter (TSC) ... *91*
10.13.1. TSC Virtualization ... 91

10.14. Supervisor Protection Keys (PKS) ... *91*

10.15. Intel® Total Memory Encryption (Intel® TME) and Multi-Key Total Memory Encryption (MKTME) ... *92*
10.15.1. TME Virtualization ... 92
10.15.2. MKTME Virtualization ... 92

10.16. Virtualization of Machine Check Capabilities and Controls ... *92*

10.17. Other Changes in TDX Non-Root Mode ... *92*
10.17.1. Tasking ... 92
10.17.2. PAUSE-Loop Exiting ... 92

11. Measurement and Attestation ... 94

11.1. TD Measurement ... *94*
11.1.1. MRTD: Build-Time Measurement Register ... 94
11.1.2. RTMR: Run-Time Measurement Registers ... 94

11.2. TD Measurement Reporting 94
11.3. TD Measurement Quoting 95
11.3.1. Intel SGX-Based Attestation 95
11.4. Quote Signing Key 96
11.5. TCB Recovery 96

12. I/O Support 98
12.1. Overview 98
12.2. Paravirtualized I/O 98
12.3. MMIO Emulation and Emulated Devices 98
12.4. Direct Device Assignment (DDA) and SRIOV 98
12.5. IOMMU – DMA Remapping 98
12.6. Shared Virtual Memory (SVM) 99

13. Intel TDX Module Lifecycle: Enumeration, Initialization and Shutdown 100
13.1. Overview 100
13.1.1. Initialization and Configuration Flow 100
13.1.2. Intel TDX Module Lifecycle State Machine 100
13.1.3. Platform Compatibility and Configuration Checking 102
13.1.3.1. Overview 102
13.1.3.2. MSR Sampling and Checks 102
13.1.3.3. CPUID Sampling, Checks and Enumeration 102
13.1.4. Physical Memory Configuration Overview 102
13.1.4.1. Intel TDX ISA Background: Convertible Memory Ranges (CMRs) 102
13.1.4.2. TDMRs and PAMT Arrays Configuration 103
13.2. Intel TDX Module Initialization Interface 105
13.2.1. Global Initialization: TDH.SYS.INIT 105
13.2.2. LP-Scope Initialization: TDH.SYS.LP.INIT 105
13.2.3. Enumeration: TDH.SYS.INFO 105
13.2.4. Global Configuration: TDH.SYS.CONFIG 105
13.2.5. Package-Scope Key Configuration: TDH.SYS.KEY.CONFIG 105
13.3. TDMR and PAMT Initialization 106
13.4. Intel TDX Module Shutdown 106
13.4.1. Shutdown Initiated by the Host VMM (as Part of Module Update) 106
13.4.2. Shutdown Initiated by a Fatal Error 106

14. Debug and Profiling Architecture 107
14.1. On-TD Debug 107
14.1.1. Overview 107
14.1.2. Generic Debug Handling 107
14.1.2.1. Context Switch 107
14.1.2.2. IA32_DEBUGCTL MSR Virtualization 107
14.1.3. Debug Feature-Specific Handling 108
14.2. On-TD Performance Monitoring 109
14.2.1. Overview 109
14.2.2. Performance Monitoring MSRs 109
14.2.3. Performance Monitoring Interrupts (PMIs) 110
14.3. Off-TD Debug 110
14.3.1. Modifying Debuggable TD's State, Controls and Memory 111
14.3.2. Preventing Guest TD Corruption of DRs 111
14.4. Uncore Performance Monitoring Interrupts (Uncore PMIs) 111

15. Memory Integrity Protection and Machine Check Handling 112

15.1. Overview 112

15.2. TDX Memory Integrity Protection Background 112
15.2.1. Cryptographic Integrity (Ci) vs. Logical Integrity (Li), MAC and TD Owner 112
15.2.2. MAC and TD Owner Update on Memory Writes 112
15.2.3. No Checks on Memory Writes 113
15.2.4. Checks on Memory Reads 113

15.3. Machine Check Architecture (MCA) Background 114
15.3.1. Uncorrected Machine Check Error 114
15.3.2. Corrected Machine Check Interrupt (CMCI) 114
15.3.3. Machine Check System Management Interrupt (MSMI) 114
15.3.4. Local Machine Check Event (LMCE) 115

15.4. Recommended MCA Platform Configuration for TDX 115

15.5. Handling Machine Check Events during Guest TD Operation 115
15.5.1. Machine Check Events Delivered as an #MC Exception (Recommended) 115
15.5.2. Machine Check Events Delivered as an MSMI (Not Recommended) 116
15.5.3. LMCE Disabled (Not Recommended) 117
15.5.4. Machine Check Events Delivered as a CMCI 117

15.6. Handling MCE during Intel TDX Module Operation 117

16. Side Channel Attack Mitigation Mechanisms 118

16.1. Checking CPU Vulnerabilities to Known Attacks 118

16.2. Branch Prediction Side Channel Attacks Mitigation Mechanisms 118

16.3. Single-Step and Zero-Step Attacks Mitigation Mechanisms 118
16.3.1. Description 118
16.3.2. Host VMM Expected Behavior 118
16.3.3. Guest TD Interface and Expected Guest TD Operation 119

17. General Aspects of the Intel TDX Interface Functions 120

17.1. Concurrency Restrictions and Enforcement 120
17.1.1. Explicit Concurrency Restrictions 120
17.1.2. Implicit Concurrency Restrictions 120
17.1.3. Transactions 121

17.2. Memory and Resource Operands Access 121
17.2.1. Overview 121
17.2.1.1. Access Semantics 121
17.2.1.2. Explicit vs. Implicit Access 122
17.2.1.3. Memory Operand Address Specification 122
17.2.1.4. Memory Type 122
17.2.1.5. Actual Memory Access vs. Memory Reference 122
17.2.1.6. Summary Table 123

17.3. Register Operands and CPU State Convention 123
17.3.1. Overview: Regular vs. Transition Leaf Functions 123
17.3.2. Interface Function Completion Status 123
17.3.2.1. Least Detailed Level: Success/Warning/Error 124
17.3.2.2. Medium Detailed Level: Class and Recoverability 124
17.3.2.3. Most Detailed Level 124
17.3.3. Other CPU State Convention 125
17.3.4. Transition Cases: TD Entry and Exit 125
17.3.4.1. TD Entry: TDH.VP.ENTER 125
17.3.4.2. TD Synchronous Exit: TDG.VP.VMCALL 125

17.4. Metadata Access Interface 126
17.4.1. Introduction 126

17.4.2. Metadata Fields and Elements 126
17.4.3. Arrays of Metadata Fields 126
17.5. Latency of the Intel TDX Interface Functions 127

SECTION 3: INTEL TDX APPLICATION BINARY INTERFACE (ABI) REFERENCE 128

18. ABI Reference: CPU Virtualization Tables 129
18.1. MSR Virtualization 129
18.2. CPUID Virtualization 132

19. ABI Reference: Constants 140
19.1. Interface Function Completion Status Codes 140
19.1.1. Function Completion Status Code Classes (Bits 47:40) 140
19.1.2. Function Completion Status Codes 140
19.1.3. Function Completion Status Operand IDs 143

20. ABI Reference: Data Types 145
20.1.1. Mutex 145
20.1.2. Shared/Exclusive (Readers/Writer) Lock 145
20.2. Basic Crypto Types 147
20.3. TD Parameters Types 147
20.3.1. ATTRIBUTES 147
20.3.2. XFAM 148
20.3.3. CPUID_VALUES 148
20.3.4. TD_PARAMS 148
20.4. Physical Memory Management Types 150
20.4.1. Physical Page Size 150
20.5. TD Private Memory Management Data Types: Secure EPT 150
20.5.1. Secure EPT Levels 151
20.5.2. Secure EPT Entry Information as Returned by TDX Module Functions 151
20.5.2.1. Returned Secure EPT Entry Content 151
20.5.2.2. Additional Returned Secure EPT Information 152
20.6. TD Entry and Exit Types 152
20.6.1. Extended Exit Qualification 152
20.7. Measurement and Attestation Types 153
20.7.1. CPUSVN 153
20.7.2. TDREPORT_STRUCT 153
20.7.3. REPORTMACSTRUCT (Reference) 154
20.7.4. REPORTTYPE (Reference) 155
20.7.5. TDINFO_STRUCT 155
20.8. Configuration, Enumeration and Initialization Types 156
20.8.1. CPUID_CONFIG 156
20.8.2. TDSYSINFO_STRUCT 156
20.8.3. CMR_INFO 159
20.8.4. TDMR_INFO 159
20.9. Metadata Access Types 160
20.9.1. MD_FIELD_ID: Metadata Field Identifier 160
20.9.2. TDR and TDCS Metadata Fields 161
20.9.3. TDVPS Metadata Field Codes 161

21. ABI Reference: Control Structures 163
21.1. TD-Scope Control Structures 163
21.1.1. How to Read the TDR and TDCS Tables 163

21.1.2. TDR 163
21.1.3. TDCS 164

21.2. TDVPS: VCPU-Scope Control Structure 166
21.2.1. Overview 166
21.2.2. How to Read the TDVPS (including TD VMCS) Tables 167
21.2.2.1. VMM Access using TDH.VP.RD and TDH.VP.WR 167
21.2.2.2. Text Highlighting 167
21.2.3. TDVPS (excluding TD VMCS) 167
21.2.4. TD VMCS 171
21.2.4.1. TD VMCS Guest State Area 172
21.2.4.2. TD VMCS Host State Area 174
21.2.4.3. TD VMCS VM-Execution Control Fields 174
21.2.4.4. TD VMCS VM-Exit Control Fields 182
21.2.4.5. TD VMCS VM-Entry Control Fields 183
21.2.4.6. TD VMCS VM-Exit Information Fields 184

22. ABI Reference: Interface Functions 186

22.1. How to Read the Interface Function Definitions 186

22.2. Host-Side (SEAMCALL) Interface Functions 186
22.2.1. SEAMCALL Instruction (Common) 186
22.2.2. TDH.MEM.PAGE.ADD Leaf 189
22.2.3. TDH.MEM.PAGE.AUG Leaf 192
22.2.4. TDH.MEM.PAGE.DEMOTE Leaf 195
22.2.5. TDH.MEM.PAGE.PROMOTE Leaf 198
22.2.6. TDH.MEM.PAGE.RELOCATE Leaf 201
22.2.7. TDH.MEM.PAGE.REMOVE Leaf 204
22.2.8. TDH.MEM.RANGE.BLOCK Leaf 207
22.2.9. TDH.MEM.RANGE.UNBLOCK Leaf 210
22.2.10. TDH.MEM.RD Leaf 213
22.2.11. TDH.MEM.SEPT.ADD Leaf 216
22.2.12. TDH.MEM.SEPT.RD Leaf 219
22.2.13. TDH.MEM.SEPT.REMOVE Leaf 222
22.2.14. TDH.MEM.TRACK Leaf 225
22.2.15. TDH.MEM.WR Leaf 227
22.2.16. TDH.MNG.ADDCX Leaf 230
22.2.17. TDH.MNG.CREATE Leaf 232
22.2.18. TDH.MNG.INIT Leaf 234
22.2.19. TDH.MNG.KEY.CONFIG Leaf 236
22.2.20. TDH.MNG.KEY.FREEID Leaf 238
22.2.21. TDH.MNG.KEY.RECLAIMID Leaf 240
22.2.22. TDH.MNG.RD Leaf 241
22.2.23. TDH.MNG.VPFLUSHDONE Leaf 243
22.2.24. TDH.MNG.WR Leaf 245
22.2.25. TDH.MR.EXTEND Leaf 247
22.2.26. TDH.MR.FINALIZE Leaf 248
22.2.27. TDH.PHYMEM.CACHE.WB Leaf 251
22.2.28. TDH.PHYMEM.PAGE.RDMD Leaf 253
22.2.29. TDH.PHYMEM.PAGE.RECLAIM Leaf 255
22.2.30. TDH.PHYMEM.PAGE.WBINVD Leaf 258
22.2.31. TDH.SYS.CONFIG Leaf 260
22.2.32. TDH.SYS.INFO Leaf 263
22.2.33. TDH.SYS.INIT Leaf 265
22.2.34. TDH.SYS.KEY.CONFIG Leaf 268
22.2.35. TDH.SYS.LP.INIT Leaf 270
22.2.36. TDH.SYS.LP.SHUTDOWN Leaf 273
22.2.37. TDH.SYS.TDMR.INIT Leaf 274
22.2.38. TDH.VP.ADDCX Leaf 276
22.2.39. TDH.VP.CREATE Leaf 278

22.2.40. TDH.VP.ENTER Leaf 280
22.2.41. TDH.VP.FLUSH Leaf 287
22.2.42. TDH.VP.INIT Leaf 289
22.2.43. TDH.VP.RD Leaf 291
22.2.44. TDH.VP.WR Leaf 293

22.3. Guest-Side (TDCALL) Interface Functions 296
22.3.1. TDCALL Instruction (Common) 296
22.3.2. TDG.MEM.PAGE.ACCEPT Leaf 298
22.3.3. TDG.MR.REPORT Leaf 300
22.3.4. TDG.MR.RTMR.EXTEND Leaf 302
22.3.5. TDG.VM.RD Leaf 304
22.3.6. TDG.VM.WR Leaf 306
22.3.7. TDG.VP.CPUIDVE.SET Leaf 308
22.3.8. TDG.VP.INFO Leaf 310
22.3.9. TDG.VP.VEINFO.GET Leaf 312
22.3.10. TDG.VP.VMCALL Leaf 314

SECTION 1: INTRODUCTION AND OVERVIEW

1. About this Document

1.1. Scope of this Document

This document describes the architecture and the external Application Binary Interface (ABI) 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 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.
- Section 3 is an Application Binary Interface (ABI) function reference for the Intel TDX module.

1.3. Glossary

Table 1.1: 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).
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.

Acronym	Full Name	New for TDX	Description
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.
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.
SEAMLDR	**SEAM Loader**	Yes	An ACM intended to load the Intel TDX module.
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.

[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
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.
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.
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 TDCS.
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.

Acronym	Full Name	New for TDX	Description
TDVPX	**Trust Domain Virtual Processor Extension**	Yes	4KB physical pages that are intended to be the non-root pages 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 1.2: 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 1.3: 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
Error Reporting through EMCA2	RAS Integration and Validation Guide for the Intel Xeon Processor – Error Reporting through EMCA Gen 2	April 2015

1.5.2. Intel TDX Public Documents

Table 1.4: 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

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.

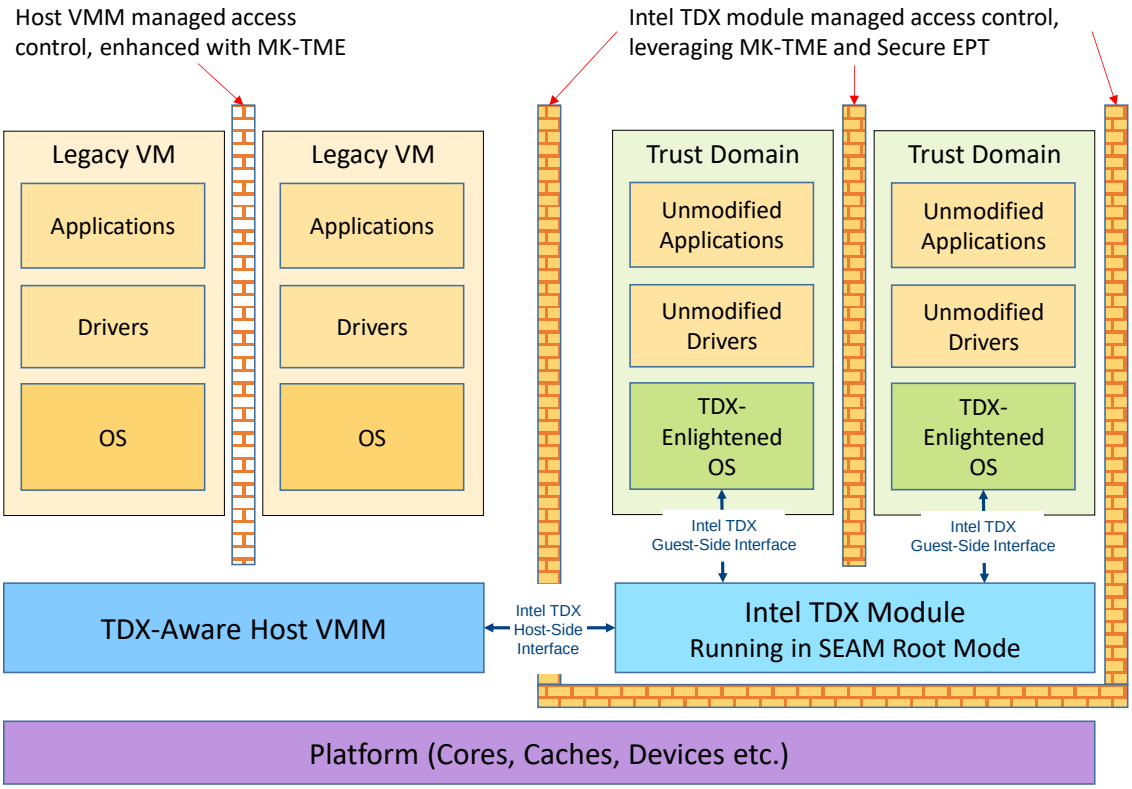

Figure 2.1: Intel® Trust Domain Extension Components Overview

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 SEAMLDR ACM.

2.1.2. 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.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 13.

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.

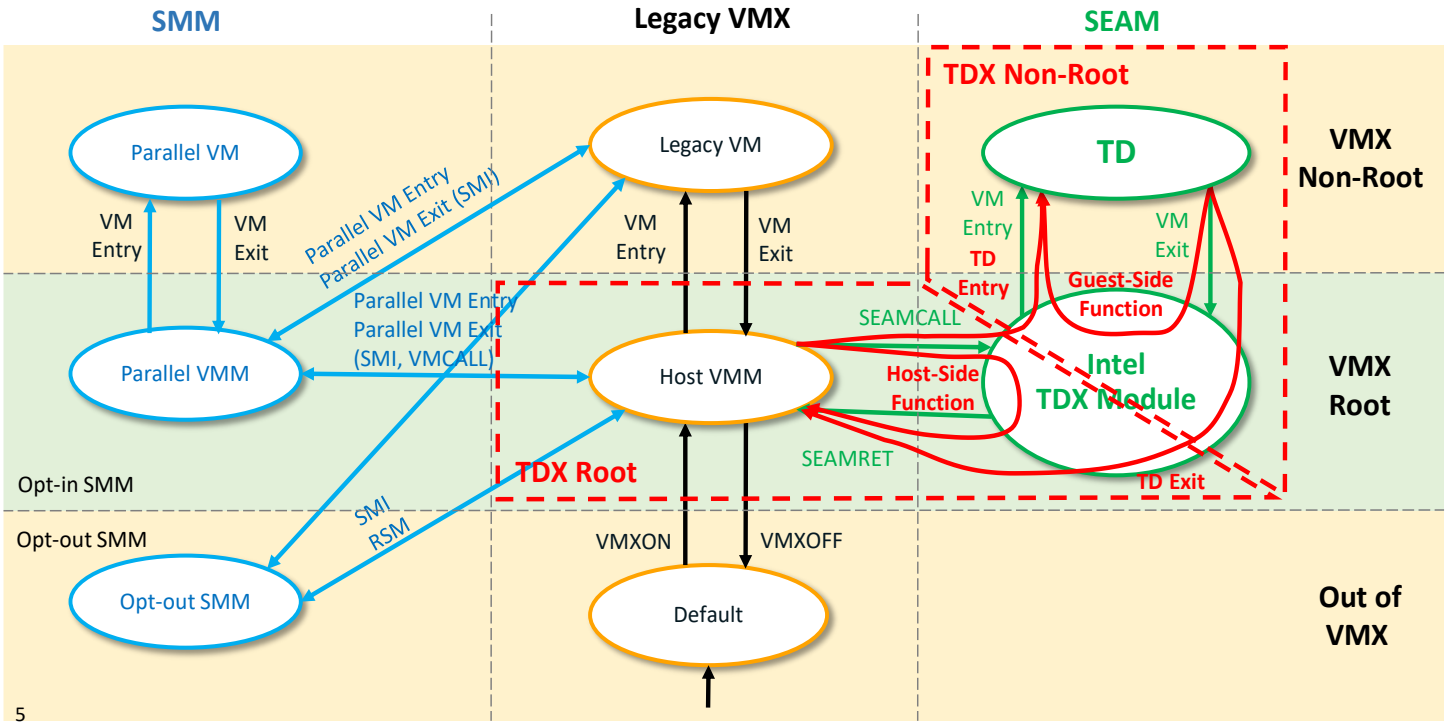

Figure 2.2: Overview of Intel TDX Modes & Transitions based on VMX and SEAM Modes and Transitions

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 ensure 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 10.

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. 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 8.

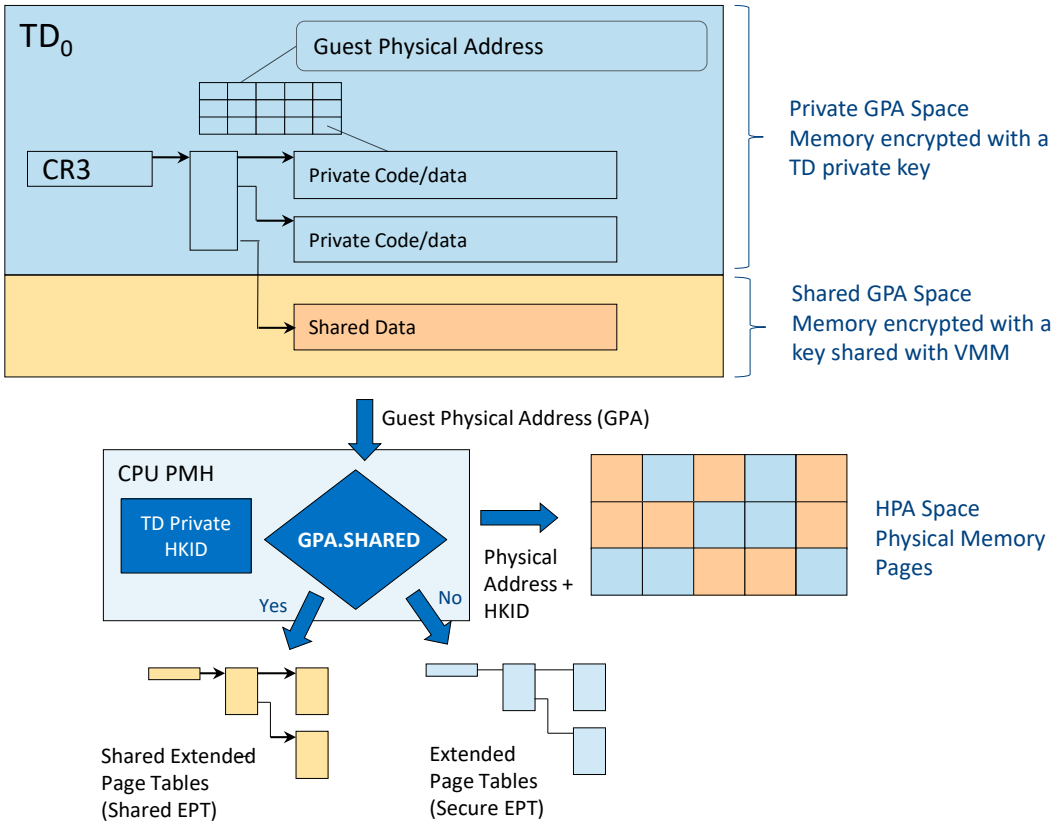

Figure 2.3: Secure EPT Concept

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 5.

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 12.

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.

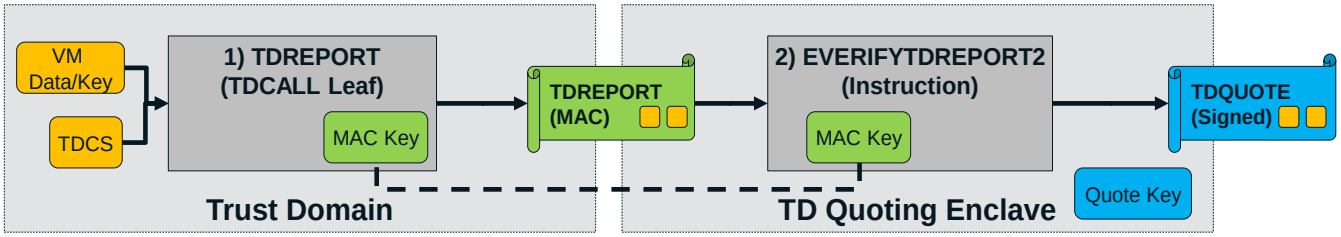

Figure 2.4: TD Attestation

TD measurement and attestation is described in Chapter 11.

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 5.

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).

2.9.1. Host-Side (SEAMCALL Leaf) Interface Functions

Table 2.3: Host-Side (SEAMCALL Leaf) Interface Functions for Intel TDX Module Management

Interface Function Name	Leaf Number	Description
TDH.SYS.CONFIG	45	Globally configure the Intel TDX module
TDH.SYS.INFO	32	Get Intel TDX module information
TDH.SYS.INIT	33	Globally initialize the Intel TDX module
TDH.SYS.KEY.CONFIG	31	Configure the Intel TDX global private key on the current package
TDH.SYS.LP.INIT	35	Initialize the Intel TDX module per logical processor
TDH.SYS.LP.SHUTDOWN	44	Shutdown the Intel TDX module on the current LP
TDH.SYS.TDMR.INIT	36	Partially initialize a Trust Domain Memory Region (TDMR)

Table 2.4: Host-Side (SEAMCALL Leaf) Interface Functions for TD Management

Interface Function Name	Leaf Number	Description
TDH.MNG.ADDCX	1	Add a control structure page to a TD
TDH.MNG.CREATE	9	Create a guest TD and its TDR root page
TDH.MNG.INIT	21	Initialize per-TD control structures
TDH.MNG.KEY.CONFIG	8	Configure the TD private key on a single package
TDH.MNG.KEY.FREEID	20	Mark the guest TD's HKID as free
TDH.MNG.KEY.RECLAIMID	27	Does nothing; provided for backward compatibility
TDH.MNG.RD	11	Read a TD-scope metadata field
TDH.MNG.VPFLUSHDONE	19	Check all of a guest TD's VCPUs have been flushed by TDH.VP.FLUSH
TDH.MNG.WR	13	Write a TD-scope metadata field

Table 2.5: VCPU-Scope Host-Side (SEAMCALL Leaf) Interface Functions

Interface Function Name	Leaf Number	Description
TDH.VP.ADDCX	4	Add a control structure page to a TD VCPU
TDH.VP.CREATE	10	Create a guest TD VCPU and its TDVPR root page
TDH.VP.ENTER	0	Enter TDX non-root operation
TDH.VP.FLUSH	18	Flush the address translation caches and cached TD VMCS associated with a TD VCPU
TDH.VP.INIT	22	Initialize the per-VCPU control structures
TDH.VP.RD	26	Read a VCPU-scope metadata field
TDH.VP.WR	43	Write a VCPU-scope metadata field

Table 2.6: Host-Side (SEAMCALL Leaf) Interface Functions for Physical Memory Management

Interface Function Name	Leaf Number	Description
TDH.PHYMEM.CACHE.WB	40	Write back the contents of the cache on a package
TDH.PHYMEM.PAGE.RDMD	24	Read the metadata of a page in a TDMR
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)
TDH.PHYMEM.PAGE.WBINVD	41	Write back and invalidate all cache lines associated with the specified memory page and HKID

Table 2.7: Host-Side (SEAMCALL Leaf) Interface Functions for TD Private Memory Management

Interface Function Name	Leaf Number	Description
TDH.MEM.PAGE.ADD	2	Add a 4KB private page to a TD during TD build time
TDH.MEM.PAGE.AUG	6	Dynamically add a 4KB private page to an initialized TD

Interface Function Name	Leaf Number	Description
TDH.MEM.PAGE.DEMOTE	15	Split a 2MB or a 1GB private TD page mapping into 512 4KB or 2MB page mappings respectively
TDH.MEM.PAGE.PROMOTE	23	Merge 512 consecutive 4KB or 2MB private TD page mappings into one 2MB or 1GB page mapping respectively
TDH.MEM.PAGE.RELOCATE	5	Relocate a 4KB mapped page from its HPA to another
TDH.MEM.PAGE.REMOVE	29	Remove a private page from a guest TD
TDH.MEM.RANGE.BLOCK	7	Block a TD private GPA range
TDH.MEM.RANGE.UNBLOCK	39	Remove the blocking of a TD private GPA range
TDH.MEM.RD	12	Read from private memory of a debuggable guest TD
TDH.MEM.SEPT.ADD	3	Add and map a 4KB Secure EPT page to a TD
TDH.MEM.SEPT.RD	25	Read a Secure EPT entry
TDH.MEM.SEPT.REMOVE	30	Remove a Secure EPT page from a TD
TDH.MEM.TRACK	38	Increment the TD's TLB tracking counter
TDH.MEM.WR	14	Write to private memory of a debuggable guest TD

Table 2.8: Host-Side (SEAMCALL Leaf) Interface Functions for TD Measurement and Attestation

Interface Function Name	Leaf Number	Description
TDH.MR.EXTEND	16	Extend the guest TD measurement register during TD build
TDH.MR.FINALIZE	17	Finalize the guest TD measurement register

2.9.2. Guest-Side (TDCALL Leaf) Interface Functions

Table 2.9: Guest-Side (TDCALL Leaf) Interface Functions

Interface Function Name	Leaf Number	Description
TDG.MEM.PAGE.ACCEPT	6	Accept a pending private page into the TD
TDG.MR.REPORT	4	Creates a cryptographic report of the TD
TDG.MR.RTMR.EXTEND	2	Extend a TD run-time measurement register.
TDG.VP.CPUIDVE.SET	5	Control delivery of #VE on CPUID instruction execution
TDG.VP.INFO	1	Get TD execution environment information
TDG.VM.RD	7	Read a TD-scope metadata field
TDG.VM.WR	8	Write a TD-scope metadata field
TDG.VP.VEINFO.GET	3	Get Virtualization Exception Information for the recent #VE exception
TDG.VP.VMCALL	0	Call a host VM service

Intel TDX interface function details are described in Chapter 22.

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. 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.
Intel TDX Module Loading	2	N/A	Platform	Any one LP	OS/VMM launches the SEAMLDR ACM, which loads the Intel TDX module.
Intel TDX Module Initialization	3	TDH.SYS.INIT	Platform	Any one LP	Perform global initialization of the Intel TDX module.
	4	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	5	TDH.SYS.INFO	Platform	Any LP	Retrieve Intel TDX module information and convertible memory (CMR) information.
	6	TDH.SYS.CONFIG	Platform	Any one LP	Configure the Intel TDX module with TDMR and PAMT setup.
	7	N/A	Package	Each Package	If any MODIFIED cache lines may exist for the PAMT ranges, flush them to memory using, e.g., WBINVD.
	8	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	9	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 Shutdown and Update

This sequence is intended to be used by the host VMM to gracefully shut down the Intel TDX module and then load a new module. All guest TDs' context and memory are lost.

Table 3.2: Typical Intel TDX Module Shutdown and Update Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
Shutdown	1	TDH.SYS.LP.SHUTDOWN	LP	Each LP	Mark the current LP as being shut down and prevent further SEAMCALLs.

Phase		Intel TDX Function	Scope	Execute On	Description
Update	2	N/A	LP	Selected LP	OS/VMM executes VMXOFF and sends INIT signal to all other LPs.
	3	N/A	LP	Other LPs	LP enters INIT state.
	4	N/A	LP	Selected LP	OS/VMM launches the SEAMLDR ACM in UPDATE scenario. SEAMLDR checks shutdown on all LPs and loads a new Intel TDX module.
	At this point, the initialization sequence continues in the same way as described in 3.1.1 above.				

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.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.	
C		1	The host VMM allocates target pages for the VCPU's TDVPR and TDVPX pages in TDMR in the context of a TD. The number of required TDVPX pages is enumerated by 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

	Step		Description	SEAMCALL Leaf Functions
	Virtual Processor Creation and Configuration (Executed per each VCPU)	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 TDVPX page, the host VMM calls the TDH.VP.ADDCX function (passing the HPA of the new TDVPX 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)*

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

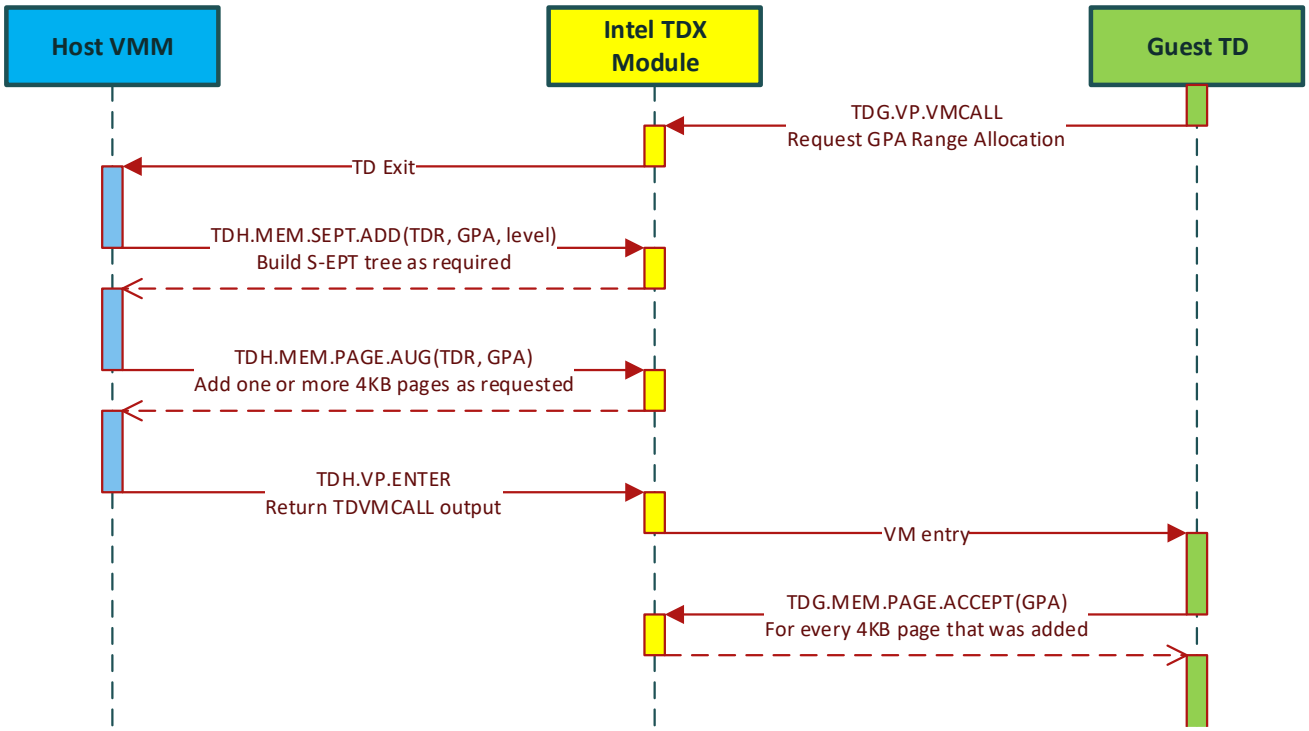

Figure 3.1: Typical Dynamic Page Addition Sequence

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.					

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 8.14.

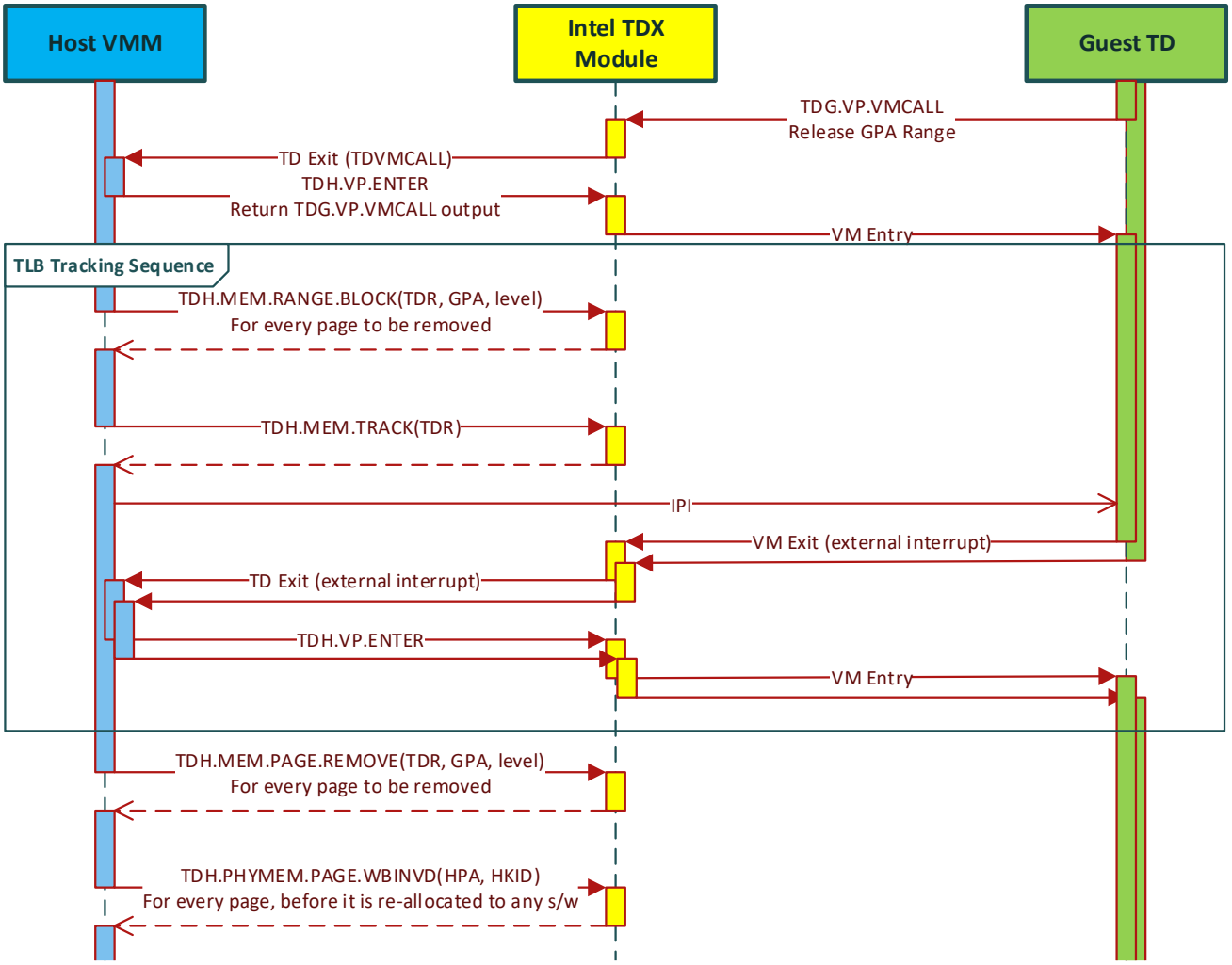

Figure 3.2: Typical Dynamic Page Removal Sequence

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 should ensure 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 8.11.

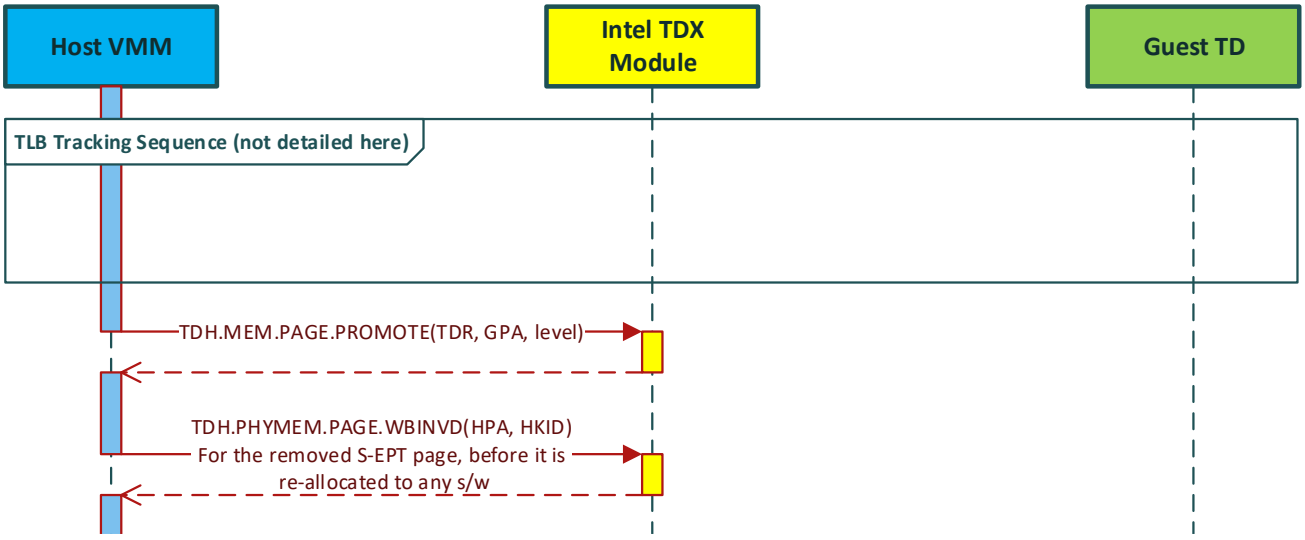

Figure 3.3: Typical Page Promotion Sequence

[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 8.12.

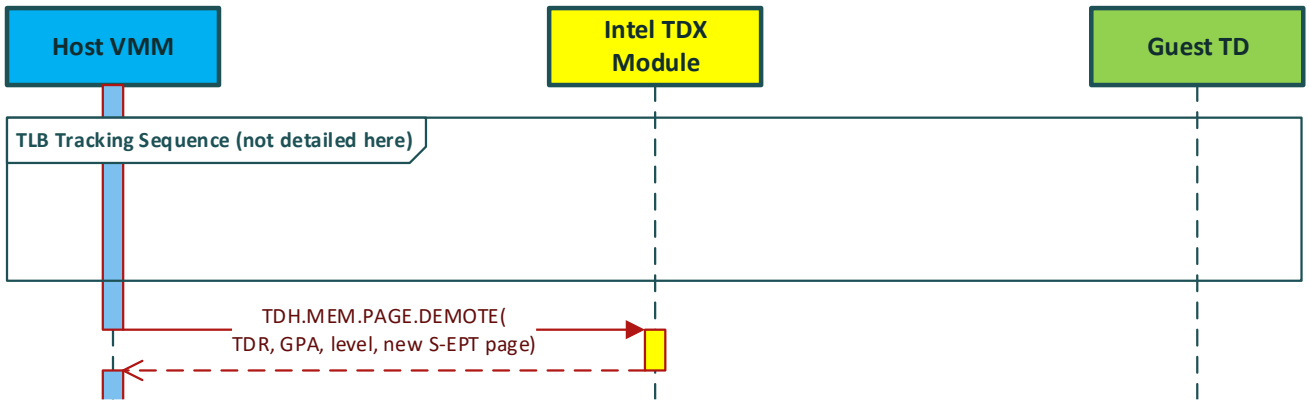

Figure 3.4: Typical Page Demotion Sequence

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 8.16.

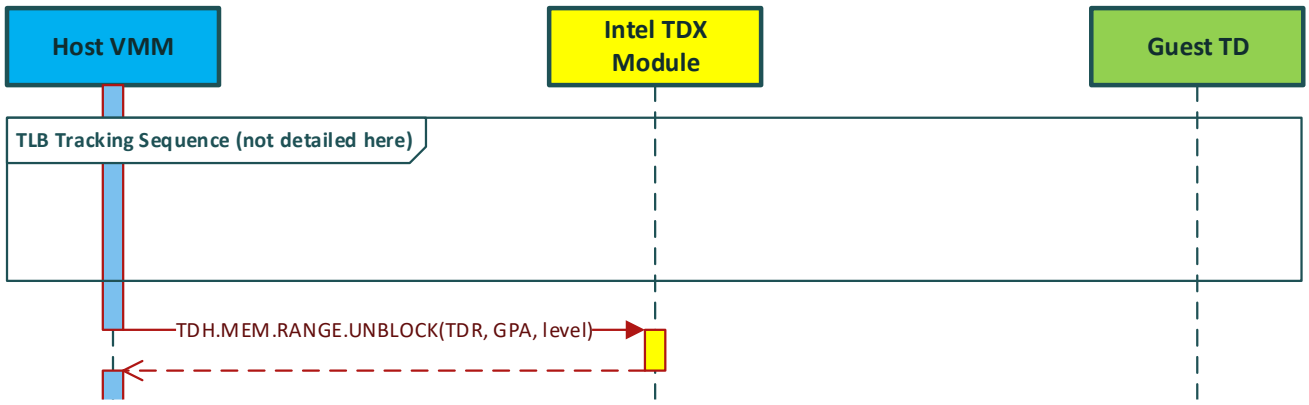

Figure 3.5: Typical GPA Range Unblock Sequence

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

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*

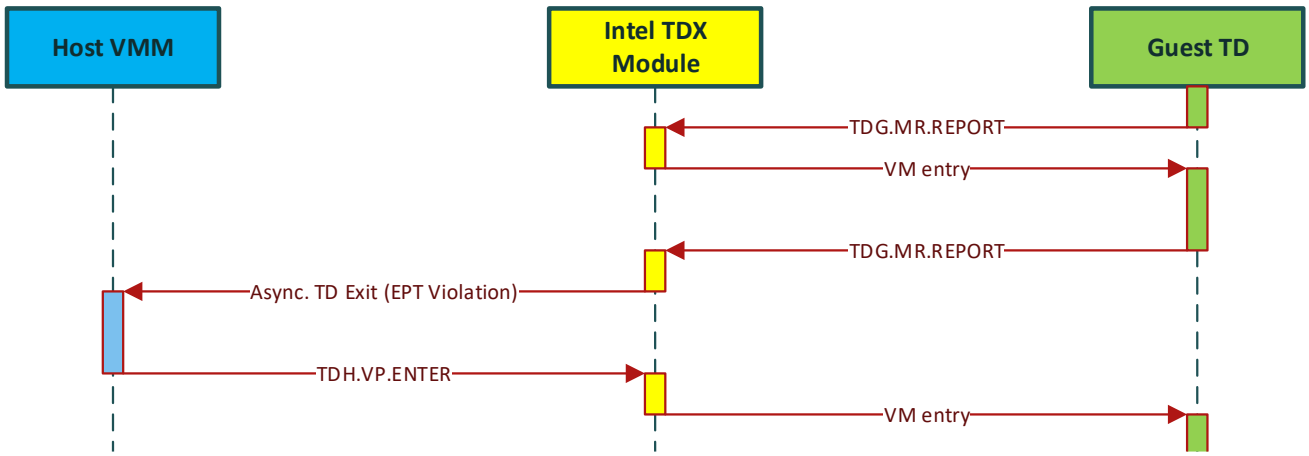

Figure 3.6: Typical Guest-Side Function Sequences

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.

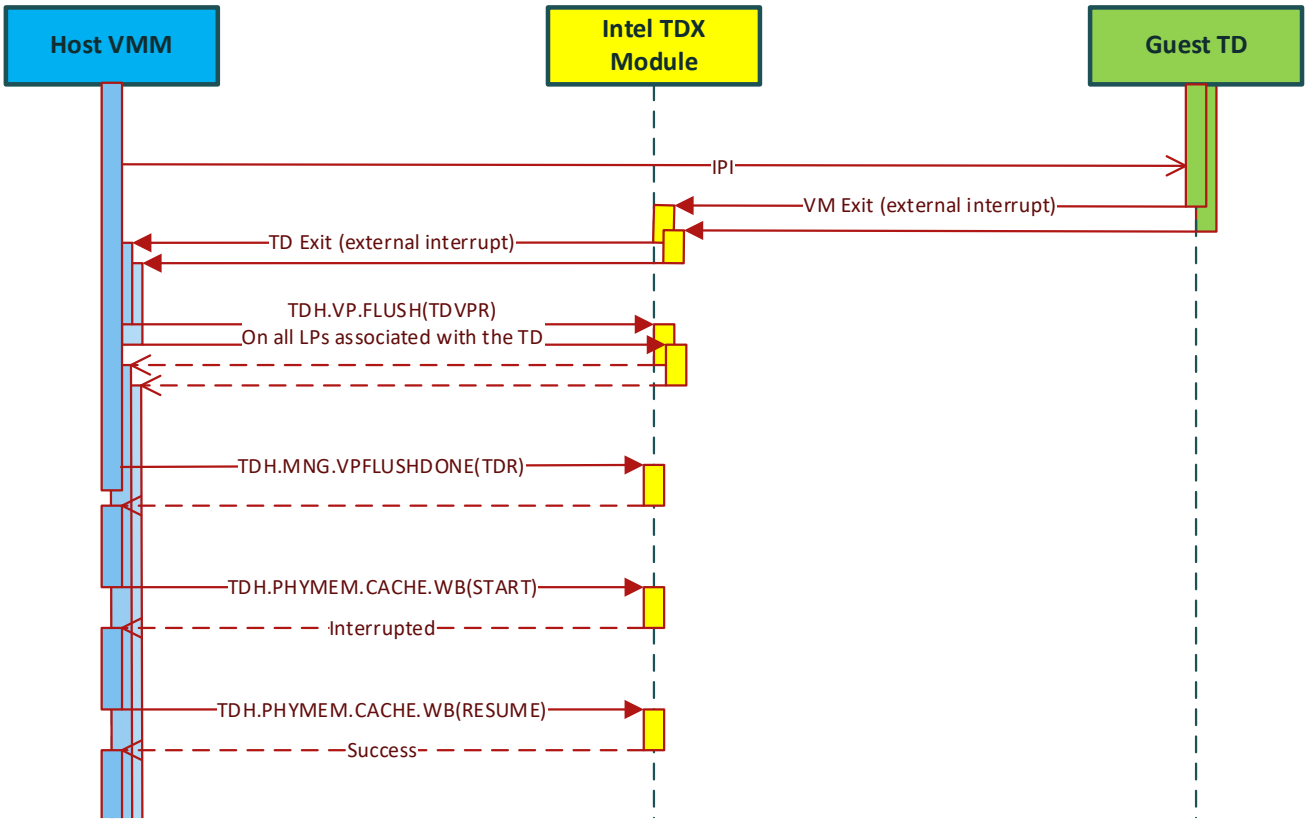

Figure 3.7: Typical TD Destruction Sequence Step A: Stopping and Flushing Out

5

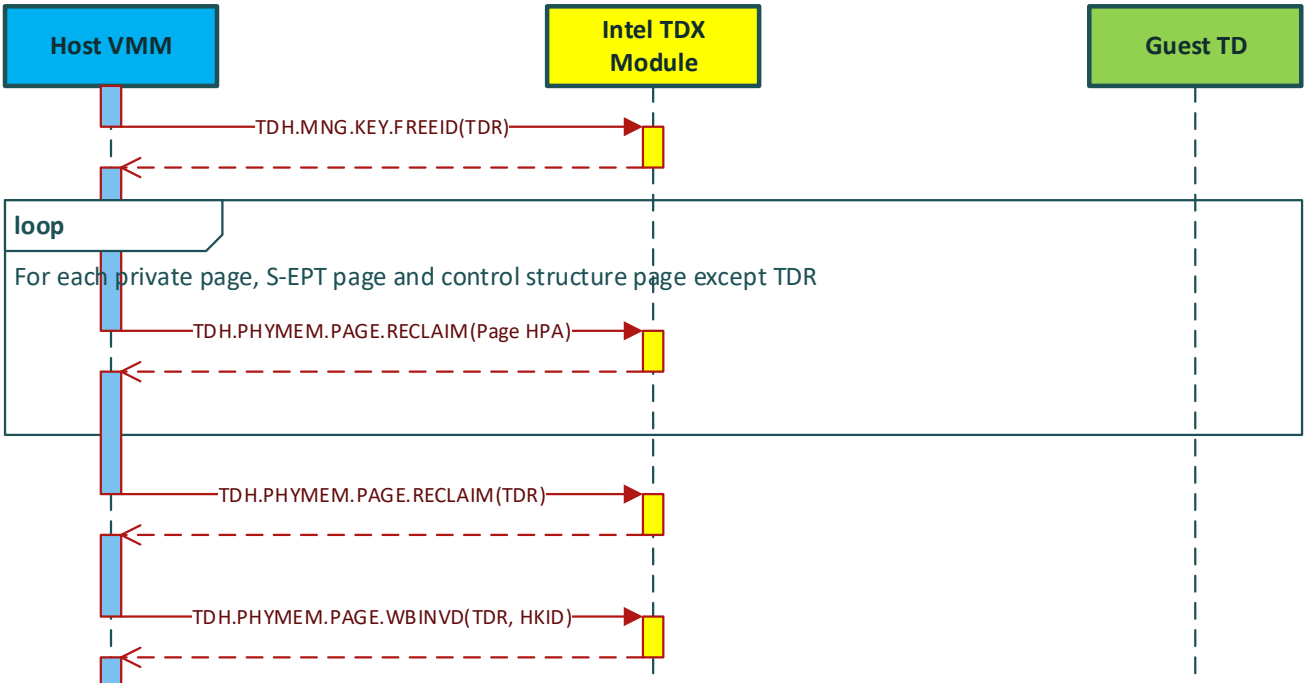

Figure 3.8: Typical TD Destruction Sequence Step B: Resource Reclamation

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. Key Management

4.1. 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 ensure that software cannot perform operations that are not compliant with TDX security objectives, as follows:

1. Help ensure 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.

4.2. 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 4.1 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 4.1: 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

4.3. 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 4.2: Key Management Tables

Table	Scope	Description	Reference
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 4.2 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.	21.1.2

Figure 4.1 below provides an abstract, high-level picture of how the tables are related. Detailed discussion is provided in the following sections.

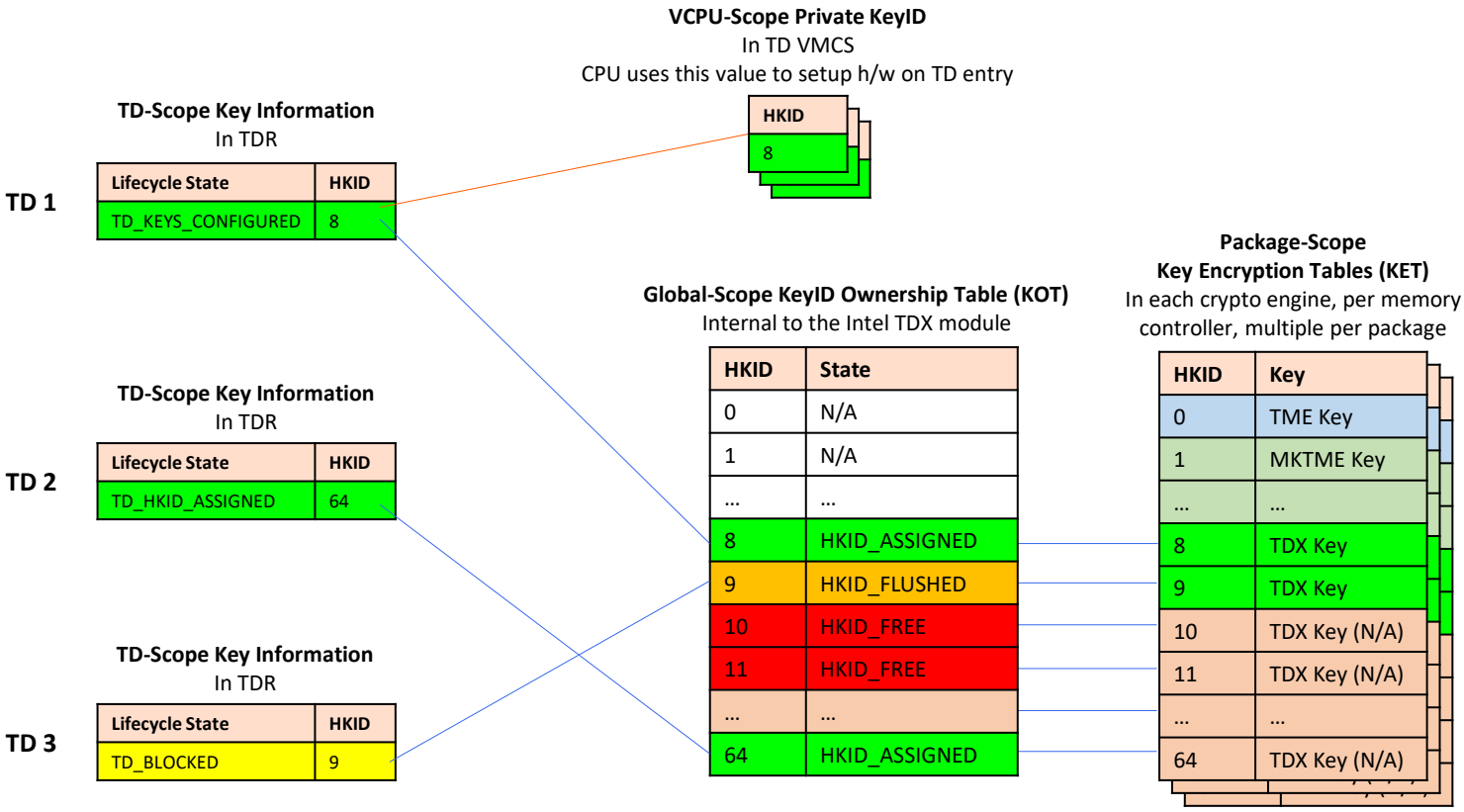

Figure 4.1: Overview of the Key Management State at TD-Scope, LP-Scope, Package-Scope and Global-Scope

4.4. 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 4.3 below lists the designed combined key management state values and their meaning. Figure 4.2 below shows a simplified diagram of the combined key state. Refer also to the key management sequences described in 4.5.

Table 4.3: 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

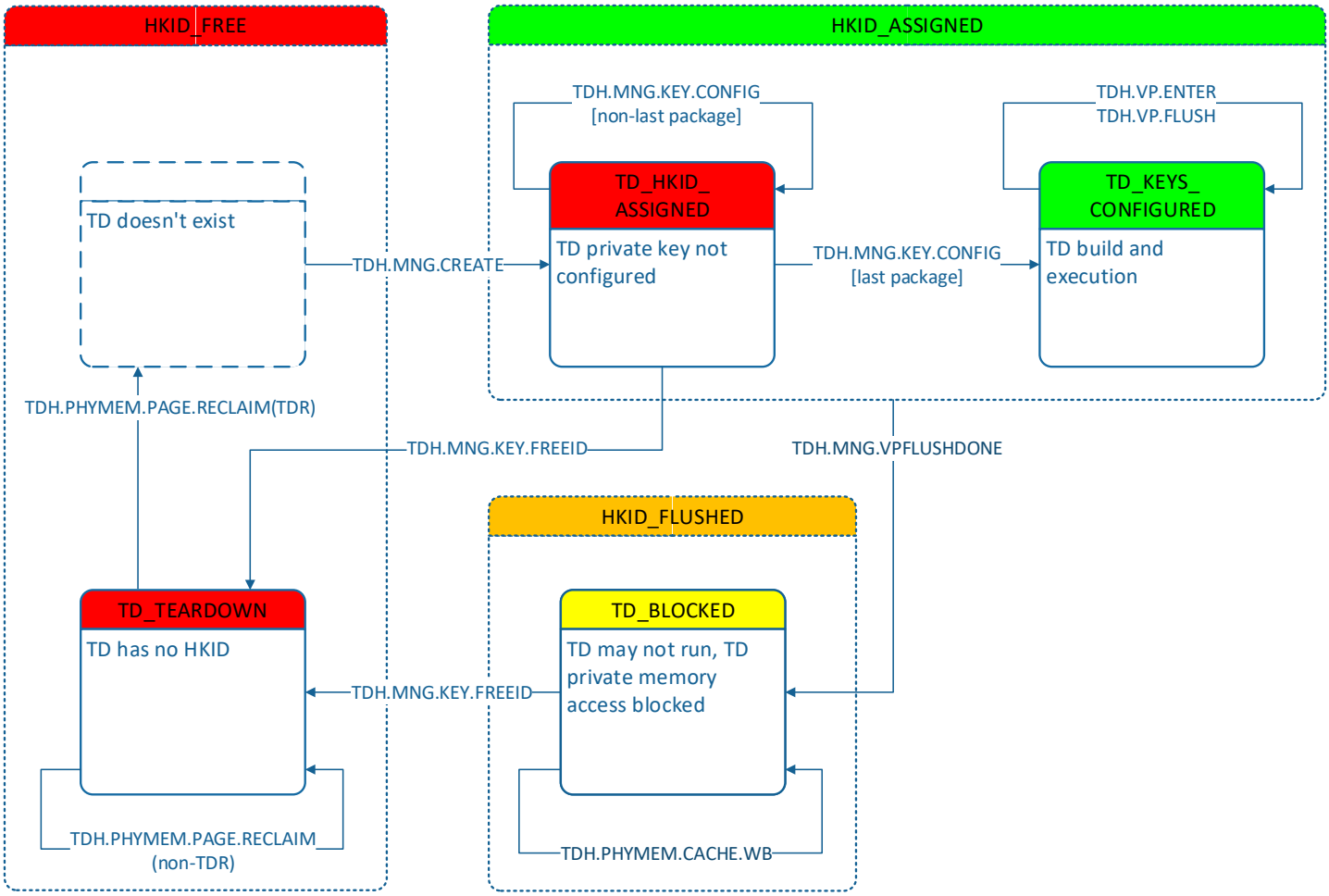

Figure 4.2: Simplified Combined TD Key Management State Diagram

Chapter 6.1 discusses TD life cycle management and zooms-in into the TD_KEYS_CONFIGURED state, detailing its secondary sub-states that control TD operation.

4.5. Key Management Sequences

4.5.1. 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 13.1.1.

4.5.2. 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 4.4: 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.

4.5.3. 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 4.5: Typical TD Keys Reclamation, 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 the TD's HKID as FREE.

[4] Enumerated by CPU during Intel TDX module initialization, see 13.1.3.3.

5. 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.

5.1. Overview

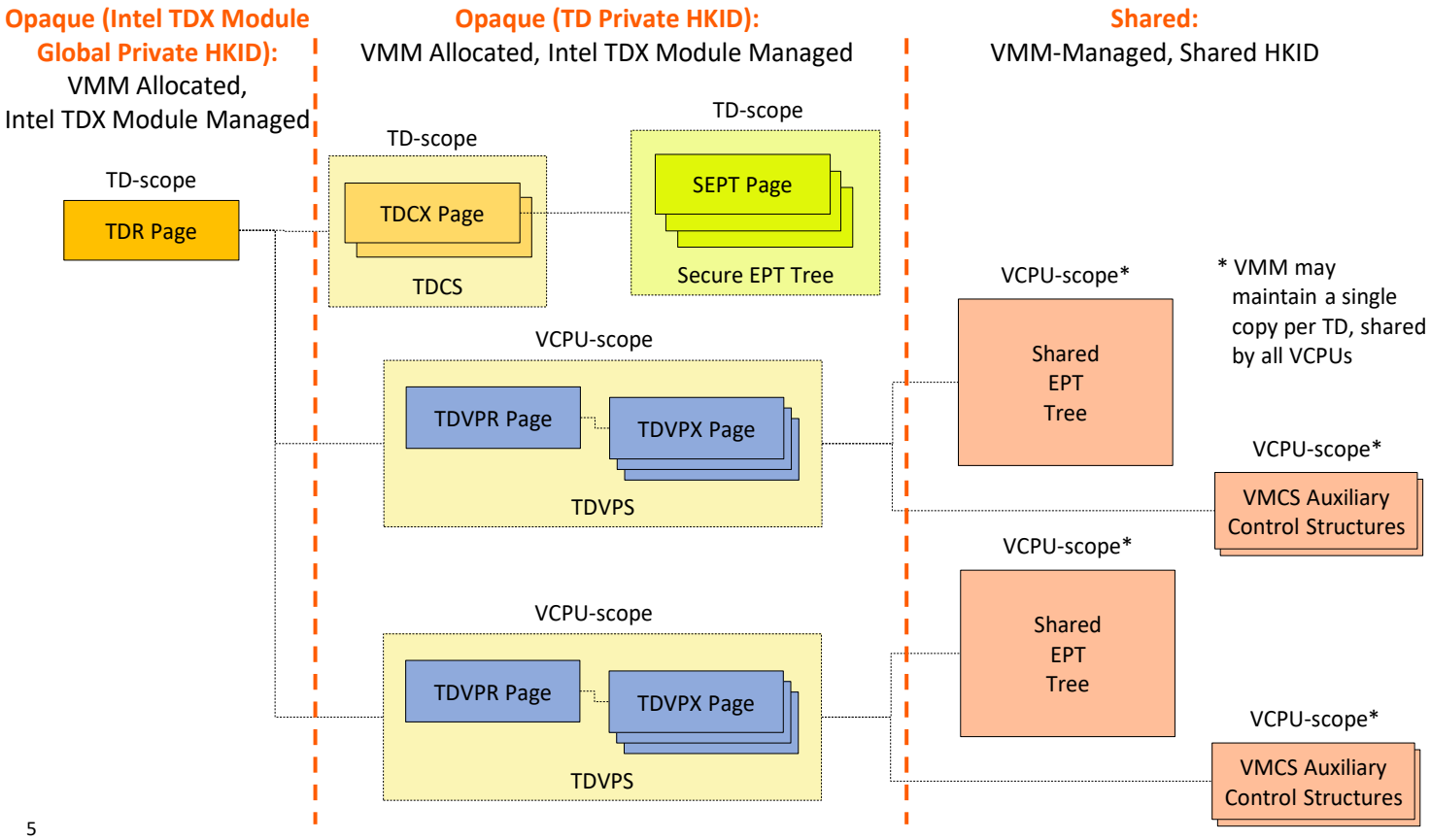

Figure 5.1: Guest TD Control Structures Overview

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.

5.1.1. 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.

5.1.2. 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.

5.2. TD-Scope Control Structures

TD-scope control structures include TDR and TDCS, discussed below, and Secure EPT, discussed in Chapter 8.

5.2.1. 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.

5.2.2. 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.

5.3. TD VCPU-Scope Control Structures and Management Functions

5.3.1. 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.

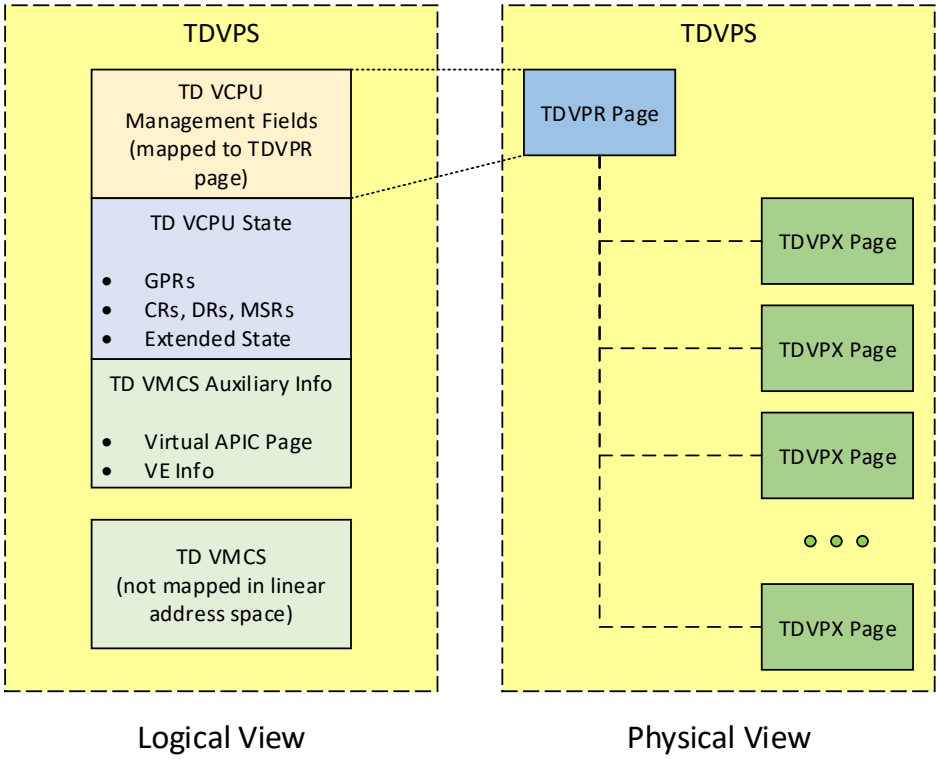

Figure 5.2: High Level Logical and Physical View of TDVPS

5.3.1.1. *Physical View of TDVPS: TDVPR/TDVPX*

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 Virtual Processor eXtension (TDVPX) 4KB pages extend TDVPR to help provide enough physical space for the logical TDVPS structure.

The TDVPR and TDVPX 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/TDVPX pages in TDVPS is enumerated to the VMM by the TDH.SYS.INFO function (see 13.2.3).

5.3.1.2. *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 21.2.

5.3.2. 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 17.2.1.1).

Non-protected control structures include:

- Shared EPT tree
- Posted interrupt descriptor

5.4. TD Non-Memory State (Metadata) Access Functions

A 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 17.4.

Table 5.1: TD Non-Memory State (Metadata) 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

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.

5.5. Concurrency Restrictions and Enforcement

A general description of concurrency restriction is provided in 17.1.

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.

6. TD Life Cycle Management

This chapter discusses guest TD life cycle management.

6.1. 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 4.4. Figure 6.1 below the TD life cycle state diagram.

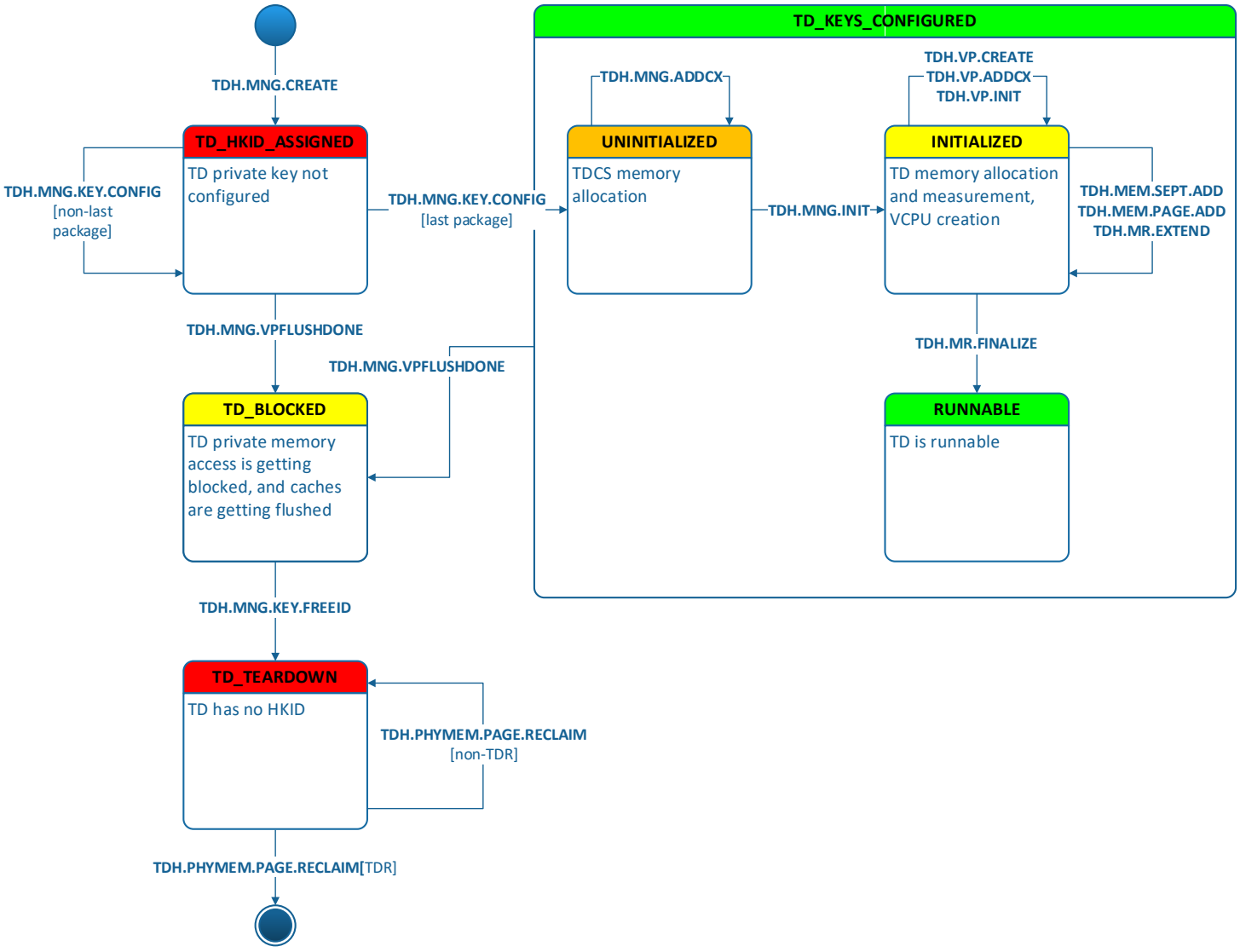

Figure 6.1: High-Level TD Life Cycle State Diagram

6.2. 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 6.1: 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 4.5.2	Assign an HKID, and configure the TD's random ephemeral key on all packages, as described in 4.5.2.
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 8. VCPUs can be created and initialized as described below.			

6.3. 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 6.2: 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, TDVPX), 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)	• TDVPX page PA • Parent TDVPR PA	Run multiple times to add the required number of TDVPX 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

6.4. 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 7.5.

Refer also to the software flow discussion in 3.4.

Table 6.3: Typical TD Teardown Sequence

	Intel TDX Function	Inputs	Description
1	Multiple	See 4.5.3	Reclaim the HKID, and flush TLB and cache, as described in 4.5.3.
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.

7. Physical Memory Management

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

7.1. 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 7.3 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.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 13.

7.2. 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.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 SEAMCALL(TDH.SYS.INFO) function, as described below.

7.3. PAMT Details

The Physical Address Metadata Table (PAMT) is designed to track the metadata of every physical page in TDMR. A page metadata include 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.

7.3.1. 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 7.3.2 below).

Table 7.1: 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 7.3 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 7.3 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 8.7 for a detailed discussion.

7.3.2. 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 three-level tree structure of **PAMT Entries**, as shown in Figure 7.1 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.

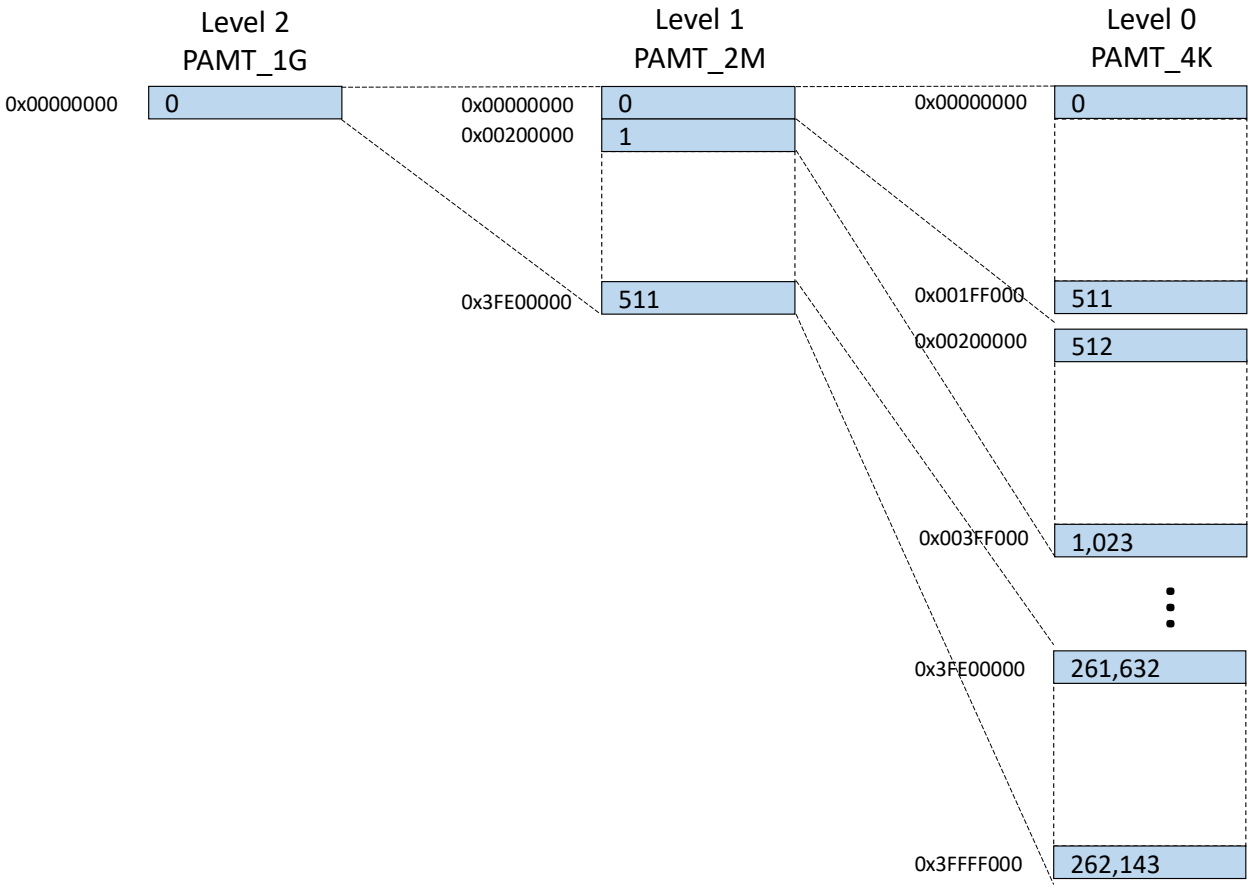

Figure 7.1: Typical Example of a PAMT Block Hierarchy for a 1GB TDMR Block

7.3.3. PAMT Hierarchy and Page Types

Table 7.2 below describes the PAMT page types:

Table 7.2: 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 7.3 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 13.1.4.2.1 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 physical page of the multi-page TDCS control structure
PT_TDVPR	PAMT_4K	TDVPR	Root page of the multi-page TDVPS control structure
PT_TDVPX	PAMT_4K	TDVPX	Non-root page of the multi-page TDVPS control structure
PT_EPT	PAMT_4K	Secure EPT	Secure EPT page

Table 7.3 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 7.3: 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_TDVPX, 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

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 8.
- PT_RSVD pages cannot be used by the Intel TDX module. They are used for implementing reserved areas within TDMRs. See 13.1.4.2.1 for details.

7.4. Adding Physical Pages

7.4.1. 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.

7.4.2. Adding Pages not Mapped to the Guest TD

By design, TD control structure pages TDR, TDCX, TDVPR and TDVPX 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, PT_TDVPR and PT_TDVPX, respectively. The overall process is described in 6.2 and 6.3.

7.4.3. 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 and TDH.MEM.PAGE.AUG add a 4KB 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 8. This section describes only their physical page management aspects.

7.5. Reclaiming Physical Pages

7.5.1. 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.

7.5.2. 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 6.3. 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.

7.5.3. 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 8; this section only highlights the physical page reclamation.

8. TD Private Memory Management

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

8.1. 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.

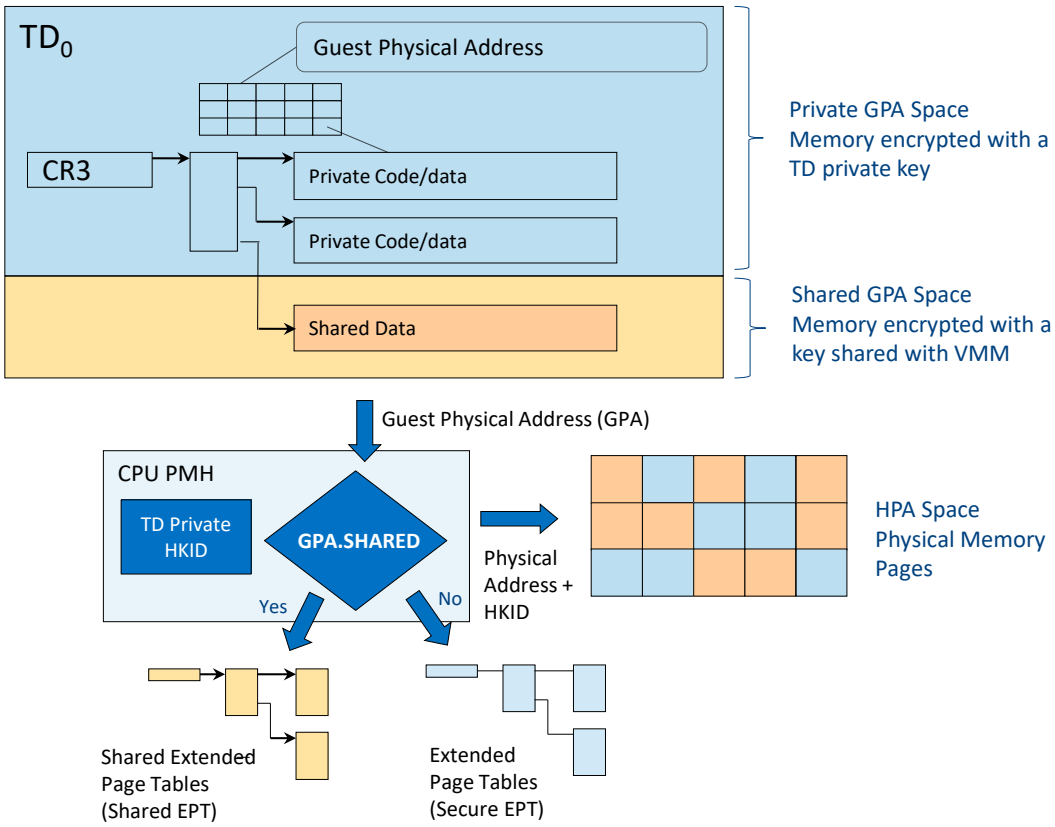

Figure 8.1: Secure EPT Concept

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.

8.2. Secure EPT Entry

8.2.1. Overview

From the CPU perspective, Secure EPT has the same structure as a legacy VMX EPT. To encode additional entry state, the Intel TDX module is designed to use two of the entry bits which are defined as available to software (not used by the CPU) when the entry is non-present. For a detailed definition of the Secure EPT structure, see 20.5.

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.

Table 8.1: Secure EPT Entry State High Level Description

Secure EPT Entry State	Present (RWX != 0)	Mapped (HPA Valid)	Description
SEPT_FREE	No	No	Secure EPT entry does not map a GPA range.
SEPT_PRESENT	Yes	Yes	Secure EPT entry maps a private GPA range which is accessible by the guest TD.
SEPT_BLOCKED	No	Yes	Secure EPT entry maps a private GPA range, but new address translations to that range are blocked.
SEPT_PENDING	No	Yes	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.
SEPT_PENDING_BLOCKED	No	Yes	Secure EPT entry is both pending and blocked.

8.2.2. SEPT Entry State Diagrams

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

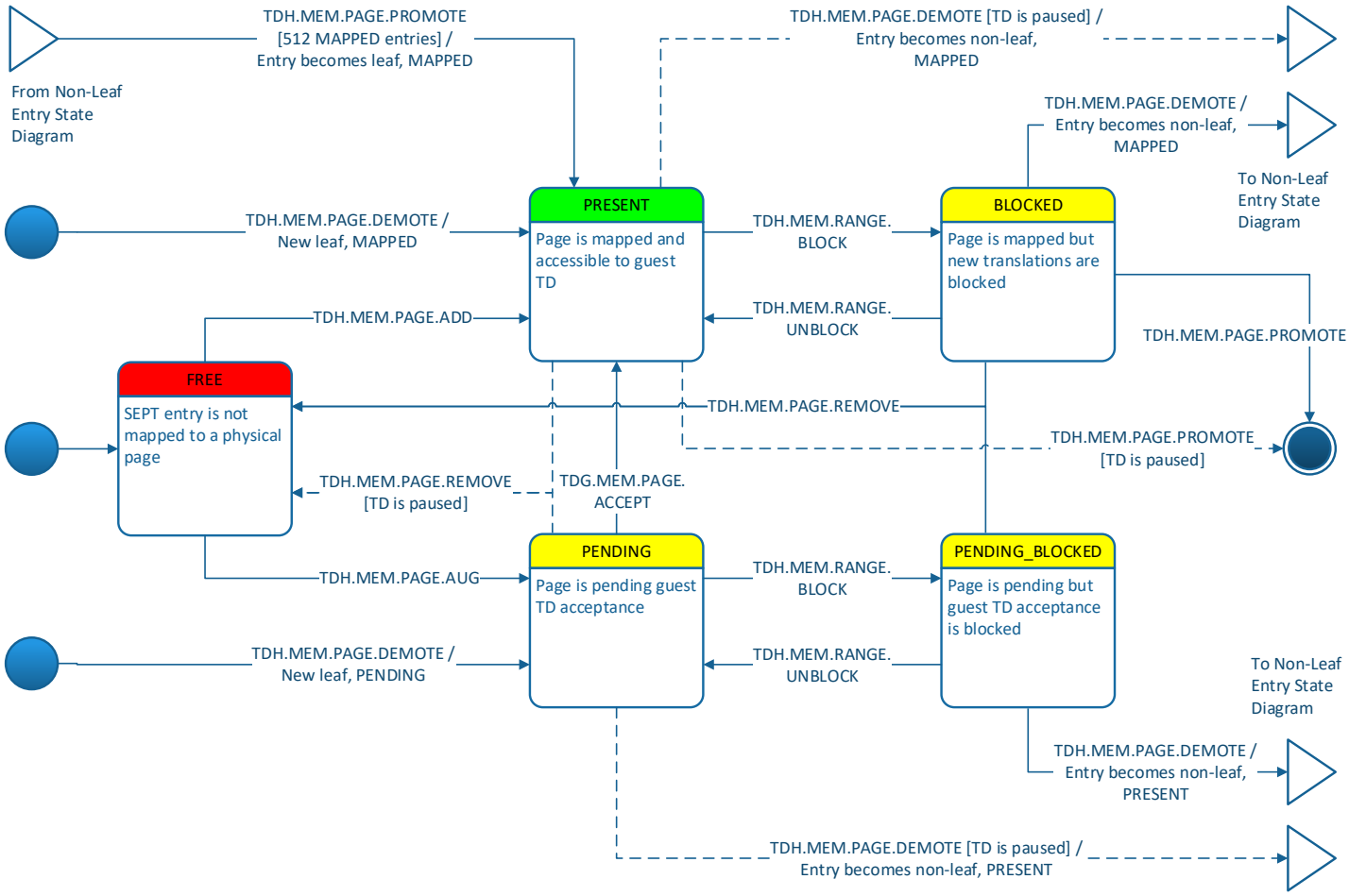

Figure 8.2: Secure EPT Leaf Entry Partial State Diagram

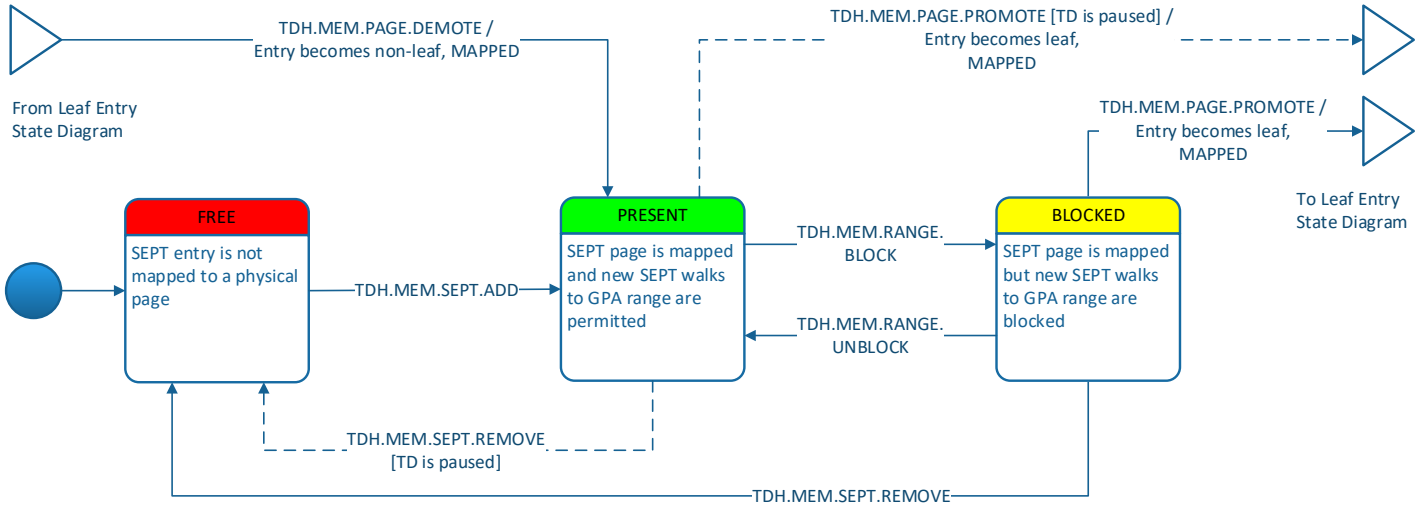

Figure 8.3: Secure EPT Non-Leaf Entry Partial State Diagram

8.3. 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. 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.

8.4. 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 8.2 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 SEPT_PRESENT state.

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.

On EPT violation TD exit, VM exit information is provided to the host VMM. This helps the VMM analyze the reason for the EPT violation and take proper action.

Table 8.2: EPT Violation TD Exit Cases and Possible Host VMM Actions

Reason	May be Indicated by	Possible Host VMM Action
Page is not mapped to the TD GPA space	• Exit qualification bits 6:3 are 0. • The host VMM knowns, based on its internal data, that either the page or a Secure EPT page that maps it has not been allocated to the TD.	The host VMM may use this as a trigger for dynamic memory allocation (TDH.MEM.PAGE.AUG) or for a post-copy migration import (see [TD Migration Spec]).
Page or GPA range is BLOCKED	• Exit qualification bits 6:3 are 0. • The host VMM knowns, based on its internal data, that the page or a Secure EPT page that maps it has been blocked.	The host VMM may resume the TD (TDH.VP.ENTER), possibly after taking some action (e.g., TDH.MEM.PAGE.PROMOTE) for which the page has been blocked.
Page is PENDING	• Exit qualification bits 6:3 are 0. • The host VMM knowns, based on its internal data, that the page has been assigned to the TD using TDH.MEM.PAGE.AUG.	This happens only if the TD's ATTRIBUTES.SEPT_VE_DISABLE is set to 1. The VMM may resume the TD.

By design, since secure EPT is fully controlled by the TDX module, an EPT misconfiguration on a private GPA indicates a TDX module bug and is handled as a fatal error.

8.5. Secure EPT Induced Exceptions

If the TD's ATTRIBUTES.SEPT_VE_DISABLE is 0, 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 10.11.1.

8.6. Secure EPT Concurrency

Secure EPT concurrency rules are designed to support the expected usage and yet be as simple as possible.

Host-Side (SEAMCALL) Functions:

- Functions that manage Secure EPT acquire **exclusive access** to the whole Secure EPT tree of the target TD.
- In specific cases where a Secure EPT entry update may collide with a concurrent update done by the guest TD, such host-side functions update the Secure EPT entry as a transaction, using atomic compare and exchange operations.
- TDH.MEM.SEPT.RD acquire **shared access** to the whole Secure EPT tree of the target TD to help prevent changes to the tree while they execute.
- Other functions that only read Secure EPT for GPA-to-HPA translation (e.g., TDH.MR.EXTEND) acquire **shared access** to the whole Secure EPT tree of the target TD to help prevent changes to the tree while they execute.

Guest-Side (TDCALL) Functions

Guest-side functions emulate the CPU's top-down EPT walk operation.

- Guest-side functions have no concurrency restrictions on the whole Secure EPT tree.
- Guest-side functions that need to update a Secure EPT entry (currently, only TDG.MEM.PAGE.ACCEPT) acquire an exclusive 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.

8.7. 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.

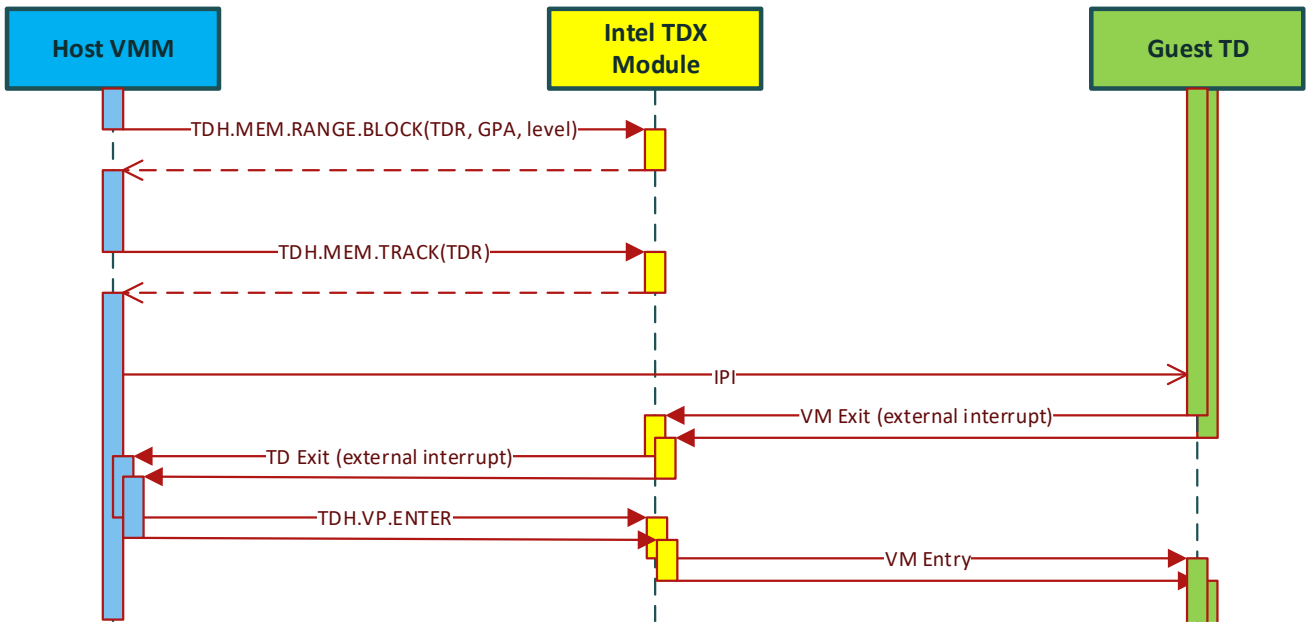

Figure 8.4: Typical TLB Tracking Sequence

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.

When each of the TD VCPUs has been inactive at least once following TDH.MEM.TRACK, 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 to each of the TD VCPUs is designed to flush them.

Note: If the host VMM counts the number of active VCPUs, and following TDH.MEM.TRACK this number is 0, the host VMM may skip the IPIs – all VCPUs are already considered tracked.

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

8.8. 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 SEPT_PRESENT.
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.

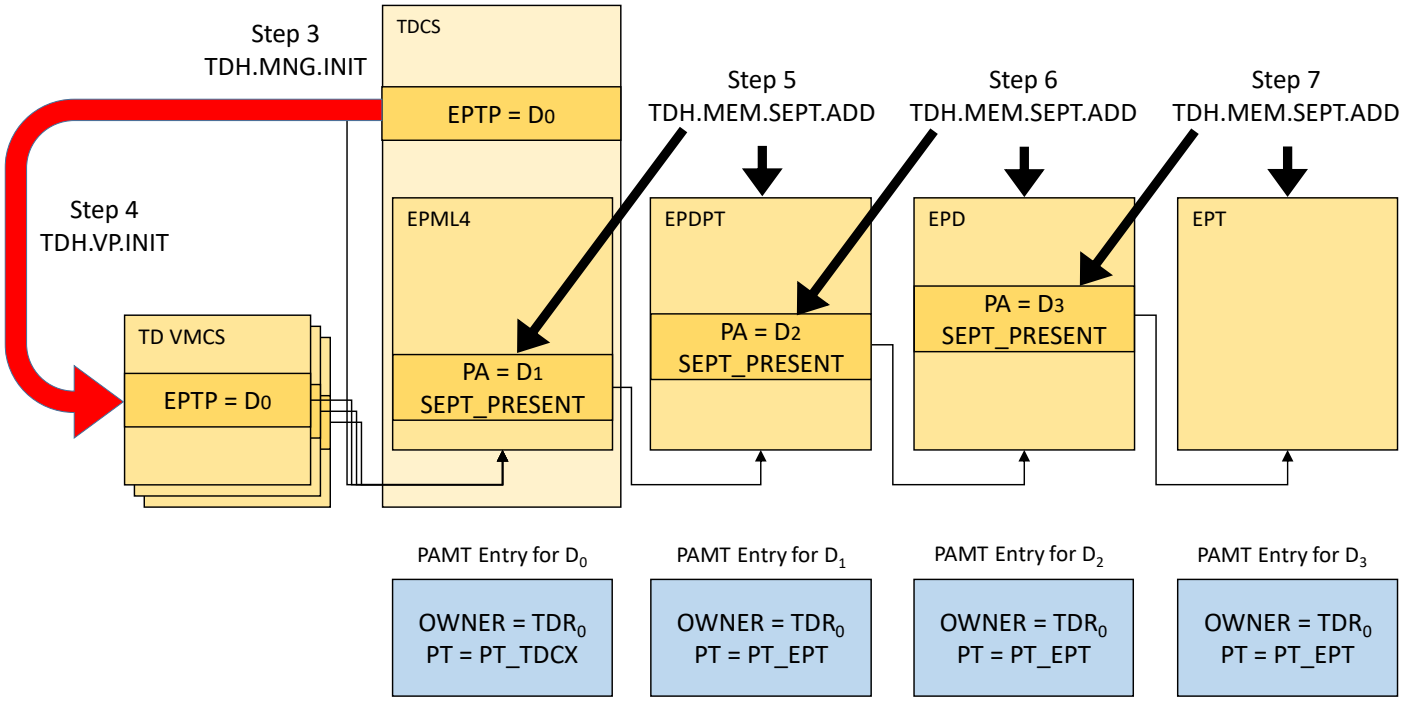

Figure 8.5: Typical Secure EPT Hierarchy Build Process

To help avoid stability issues caused by cache line aliasing, the VMM should ensure 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.

[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 ?).

8.9. 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.
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 SEPT_PRESENT.
7. Update the TD measurement with the new page GPA (as described in 11.1.1).
8. Update the PAMT entry with the PT_REG page type and the TDR PA as the OWNER.

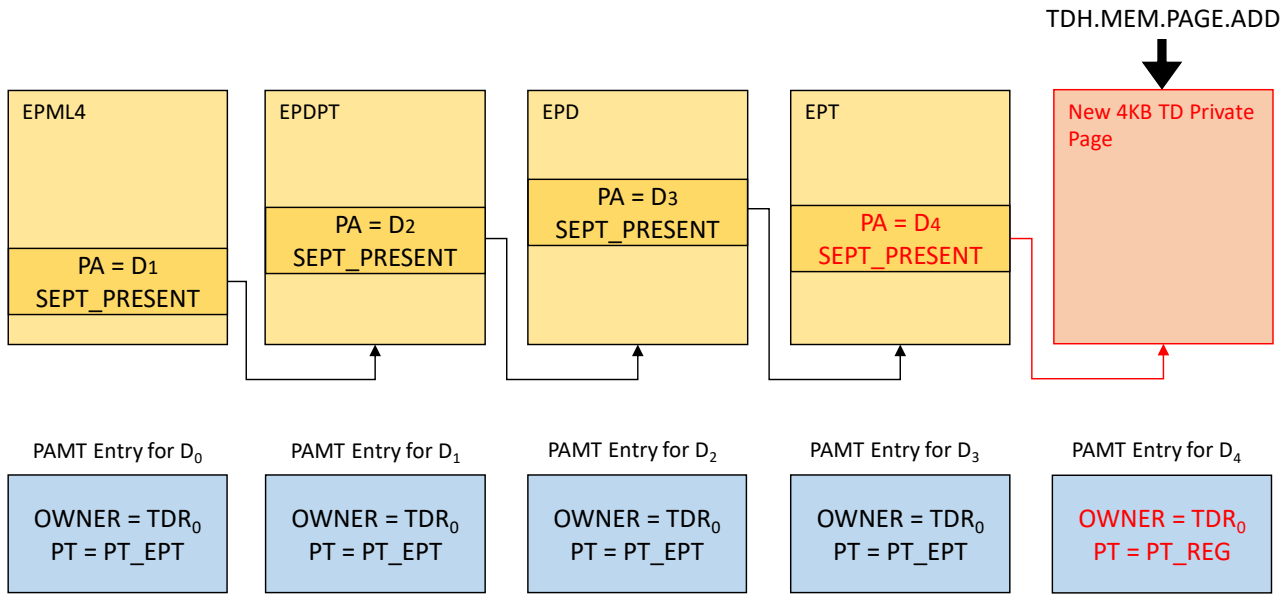

Figure 8.6: Typical Sequence for Adding a TD Private Page during TD Build Time

To help avoid stability issues caused by cache line aliasing, the VMM should ensure 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.

8.10. Dynamically Adding TD Private Pages

8.10.1. 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 SEPT_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.

8.10.2. 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 the TD has been initialized by TDH.MNG.INIT.
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 SEPT_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.

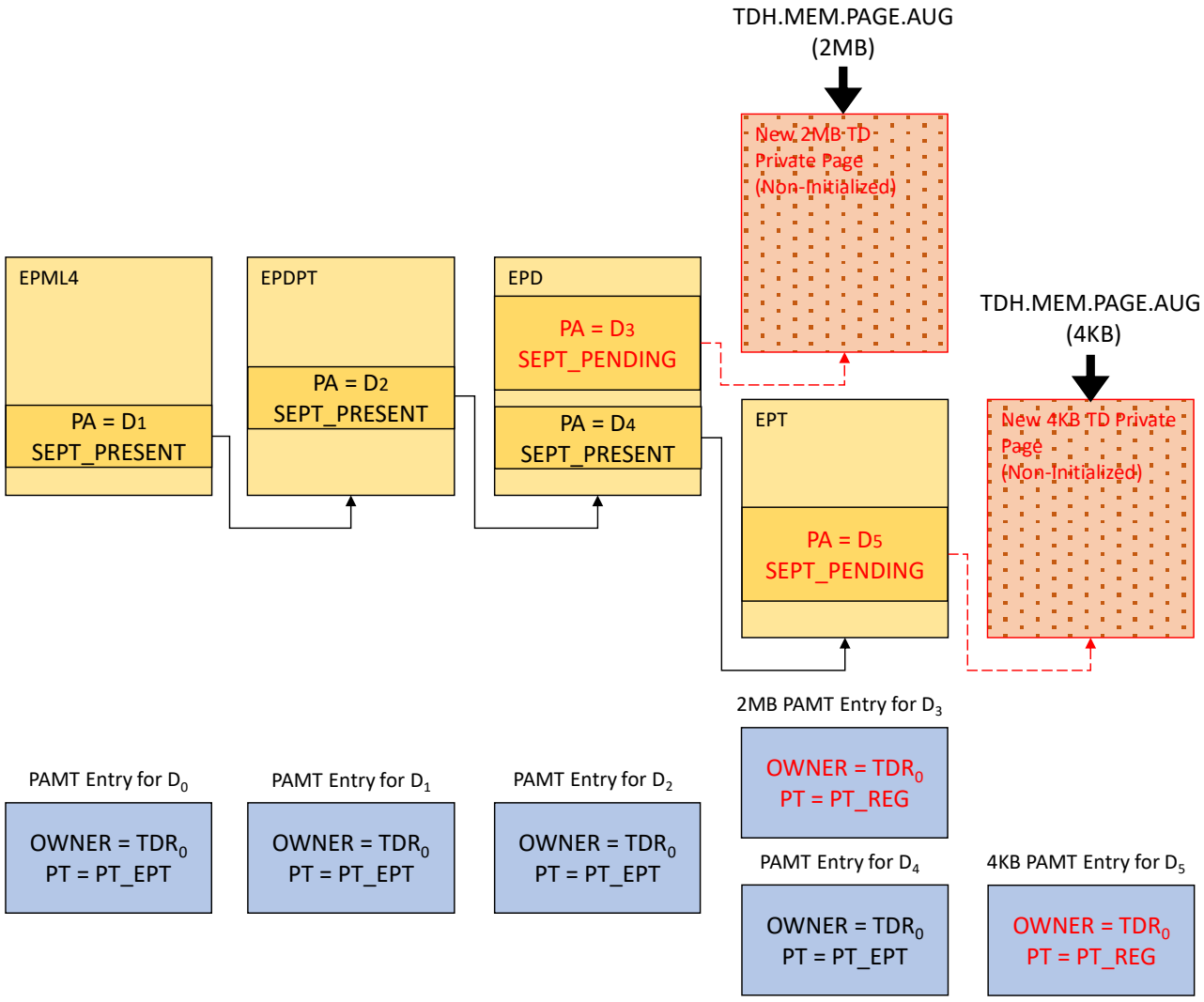

Figure 8.7: Host VMM Adding a 4KB or a 2MB TD Private Page

To help avoid stability issues caused by cache line aliasing, the VMM should ensure 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.

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

8.10.3.1. *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 8.3: 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	SEPT_PRESENT (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.
		Other than SEPT_PRESENT (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 20.6.1.	The host VMM demotes the page to match the requested accept size.
	Non-Leaf (incl. SEPT_FREE)	Other than SEPT_PRESENT (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 20.6.1.	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 (incl. SEPT_FREE)	SEPT_FREE	TD exit with EPT violation indicating the error SEPT entry level and state, and the guest-requested accept level. See 20.6.1.	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.
		Other than SEPT_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	SEPT_PRESENT	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.

SEPT Walk Terminal Entry			TDG.MEM.PAGE.ACCEPT Operation	Typical Software Handling
Level	Leaf or Non-Leaf	State		
		SEPT_BLOCKED or SEPT_PENDING_BLOCKED	TD exit with EPT violation indicating the error SEPT entry level and state, and the guest-requested accept level. See 20.6.1.	The host VMM resolves the blocking (e.g., completes the memory management operation that required blocking) and resumes the guest.
		SEPT_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.

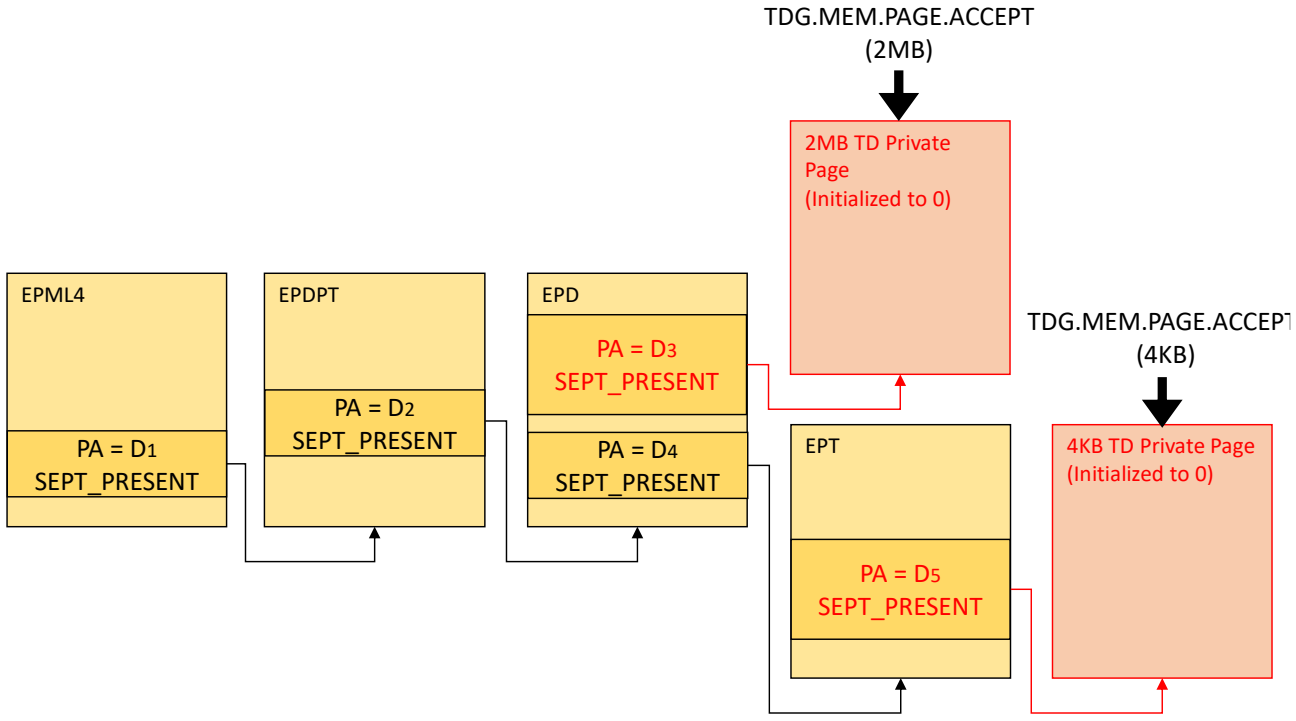

Figure 8.8: Guest TD Accepting a 4KB or 2MB Pending TD Private Page

8.10.3.2. *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 22.2.31.

8.11. 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 **SEPT_BLOCKED** and records the TD epoch in the PAMT entry of the EPT page. Figure 8.9 below shows the situation after TDH.MEM.RANGE.BLOCK blocked a 2MB GPA range.

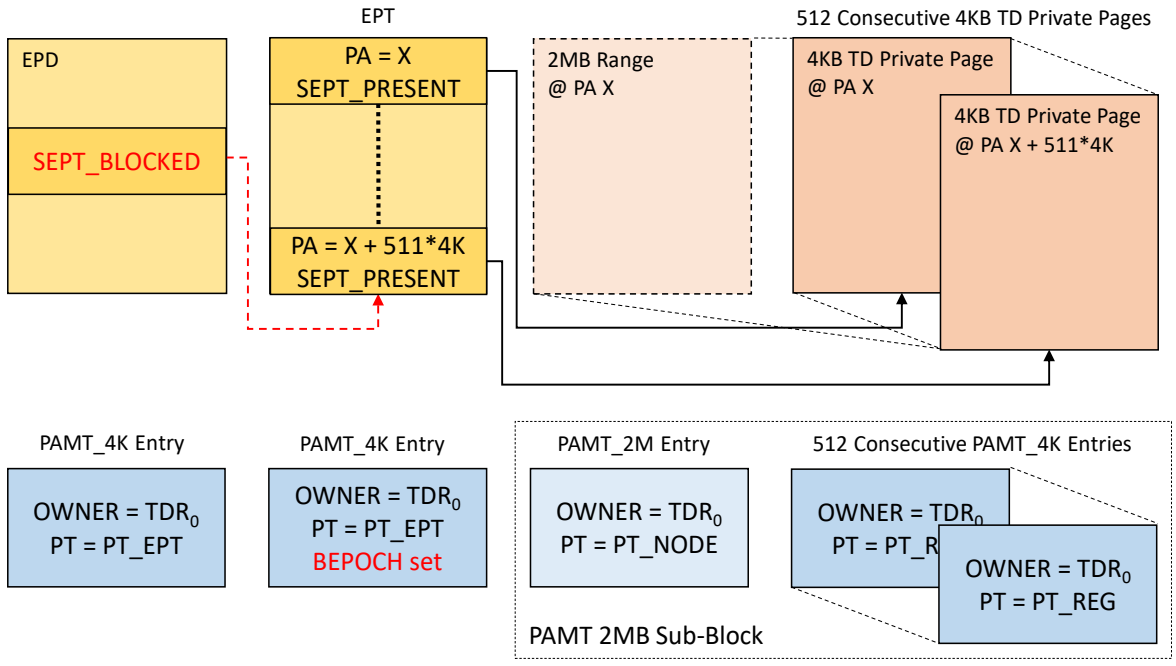

Figure 8.9: Typical State after Blocking a Range of 512 Consecutive 4KB TD Private Pages

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 SEPT_PRESENT 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 ensure that no cache lines associated with the page are in a Modified state. This is done be calling TDH.PHYMEM.PAGE.WBINVD.

Figure 8.10 below shows a typical 2MB merged page after TDH.MEM.PAGE.PROMOTE.

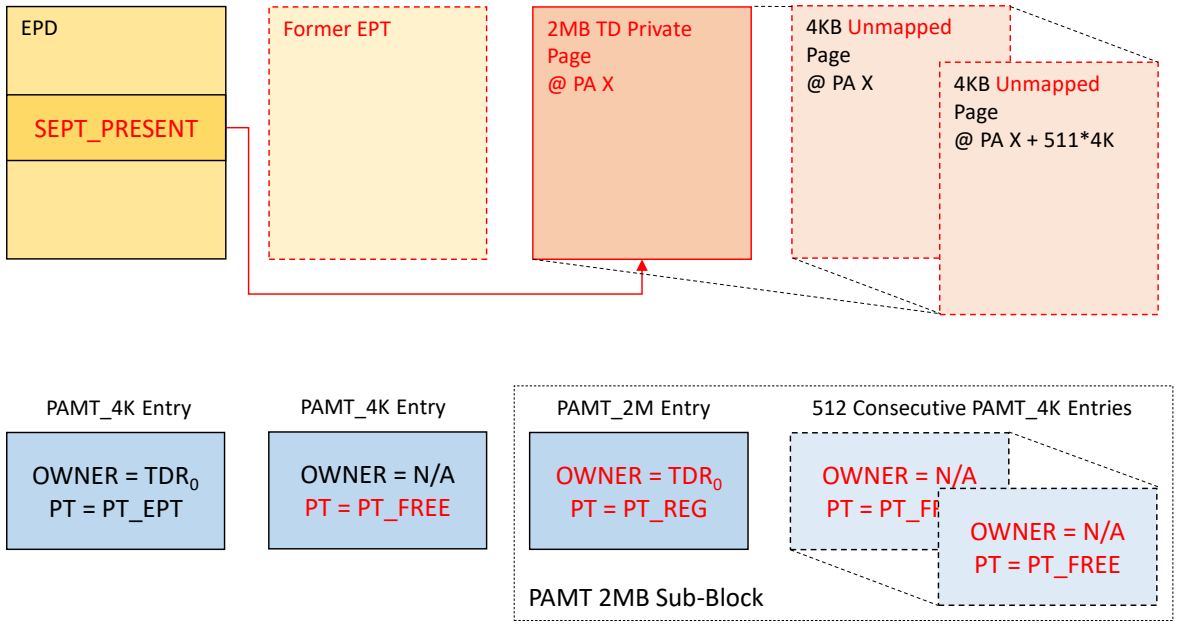

Figure 8.10: Typical State of a 2MB TD Private Page after TDH.MEM.PAGE.PROMOTE

Refer also to the software flow described in 3.3.1.3.

8.12. 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 **SEPT_BLOCKED** and records the TD epoch in the PAMT entry of the page. Figure 8.11 below shows the typical situation after TDH.MEM.RANGE.BLOCK blocked a 1GB large page.

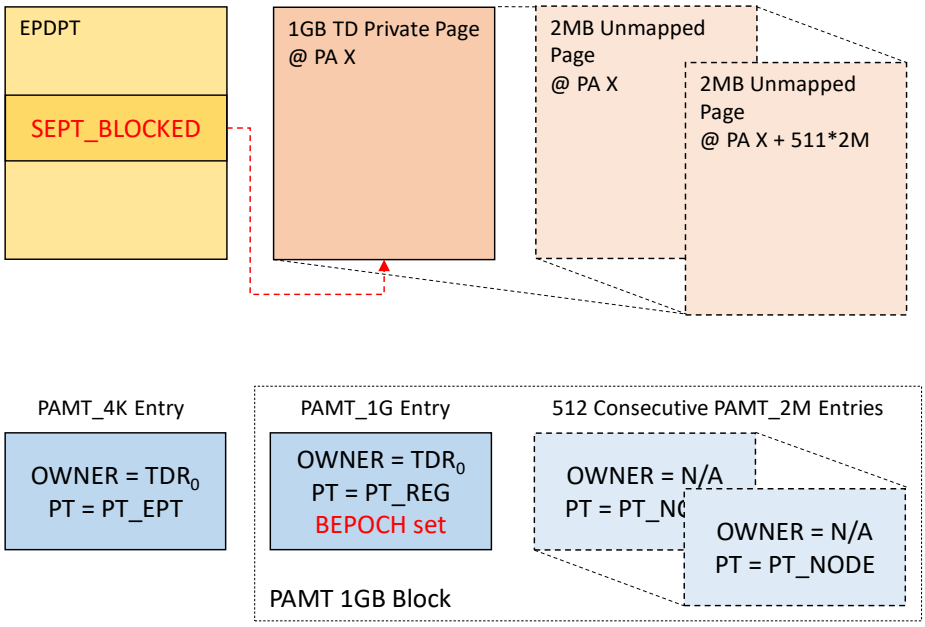

Figure 8.11: Typical State after Blocking a 1GB Page

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

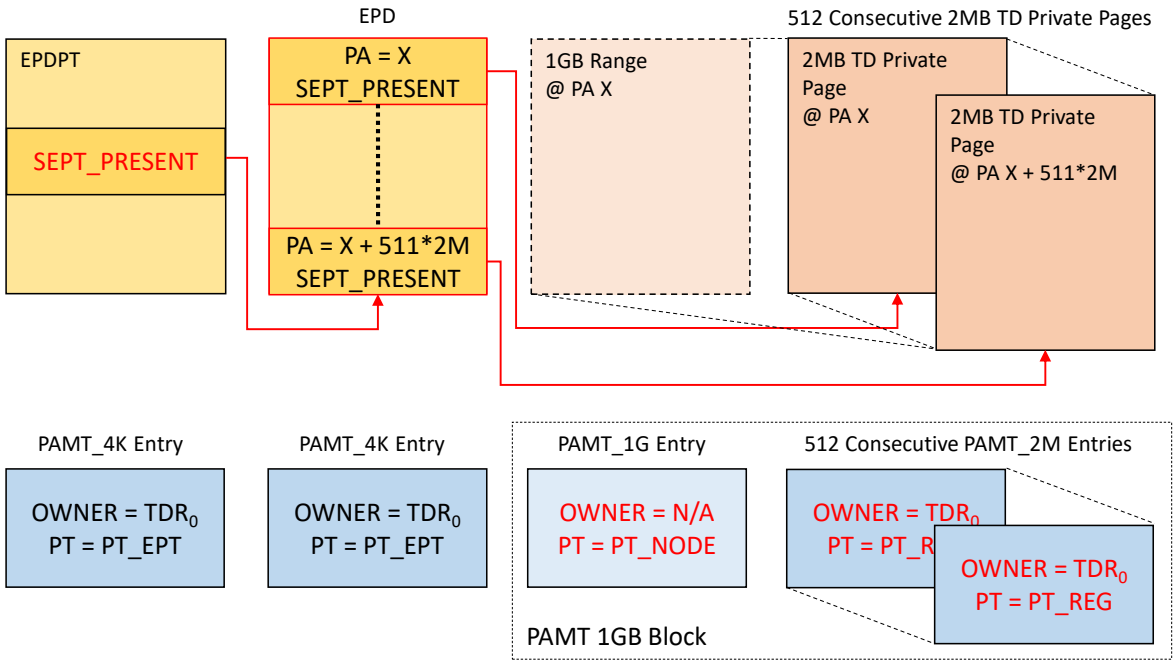

Figure 8.12: Typical State of a 1GB TD Private Range after TDH.MEM.PAGE.DEMOTE

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

Refer also to the software flow described in 3.3.1.4.

8.13. 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 **SEPT_BLOCKED** (if it was SEPT_PRESENT) or **SEPT_PENDING_BLOCKED** (if it was SEPT_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 SEPT_PRESENT or SEPT_PENDING depending on whether its previous state was SEPT_BLOCKED or SEPT_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 ensure that no cache lines associated with the old page are in a Modified state. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

8.14. 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 **SEPT_BLOCKED** (if it was SEPT_PRESENT) or **SEPT_PENDING_BLOCKED** (if it was SEPT_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 SEPT_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 ensure 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.

8.15. 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 SEPT_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 **SEPT_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 SEPT_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 ensure that no cache lines associated with the page are in a Modified state. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

8.16. 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 SEPT_PRESENT (if it was SEPT_BLOCKED) or SEPT_PENDING (if it was SEPT_PENDING_BLOCKED).

Refer also to the software flow described in 3.3.1.5.

9. TD VCPU

This chapter discusses multiple items related to TD VCPUs.

9.1. VCPU Transitions

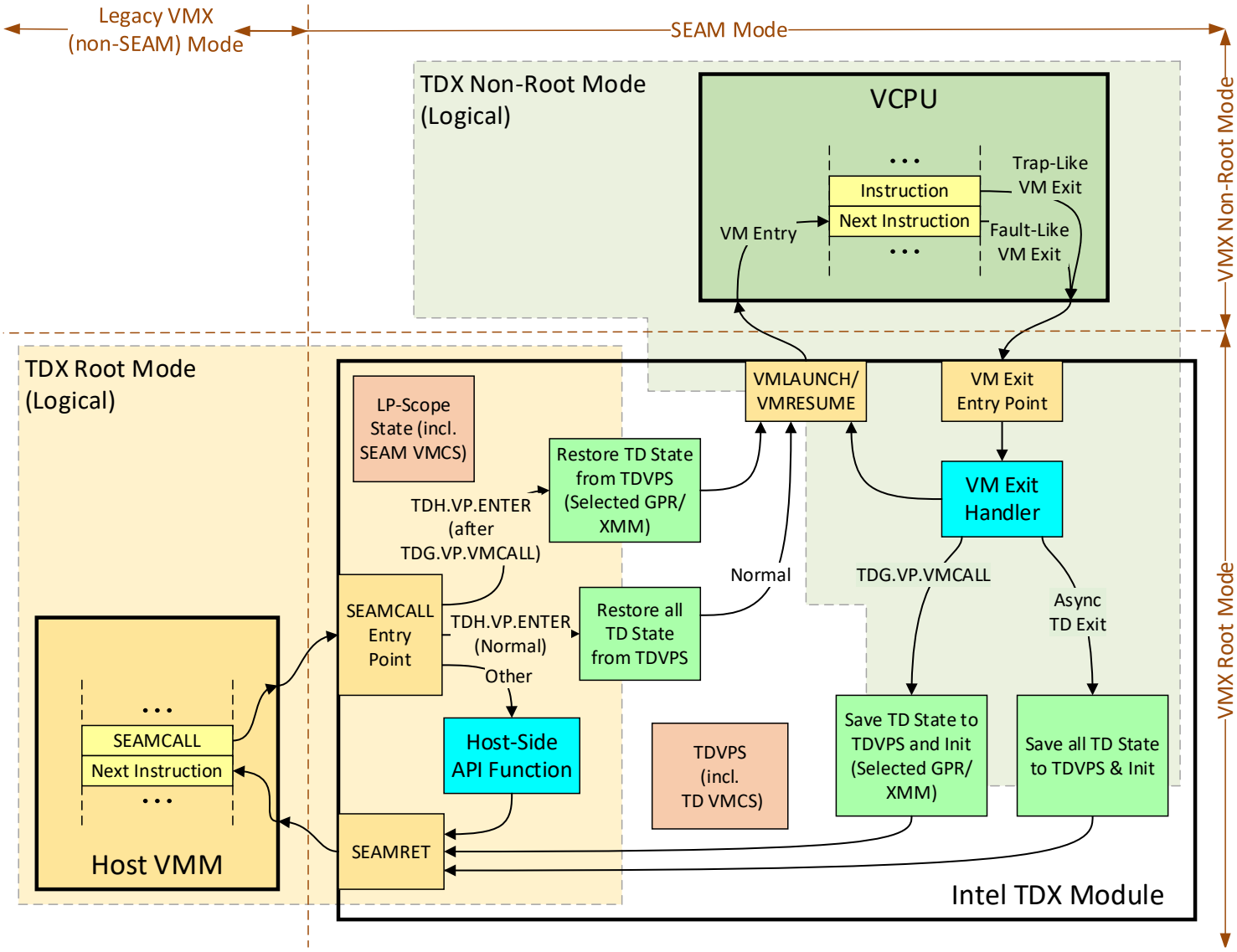

Figure 9.1: TD VCPU Transitions Overview

9.1.1. 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 22.2.40.

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.

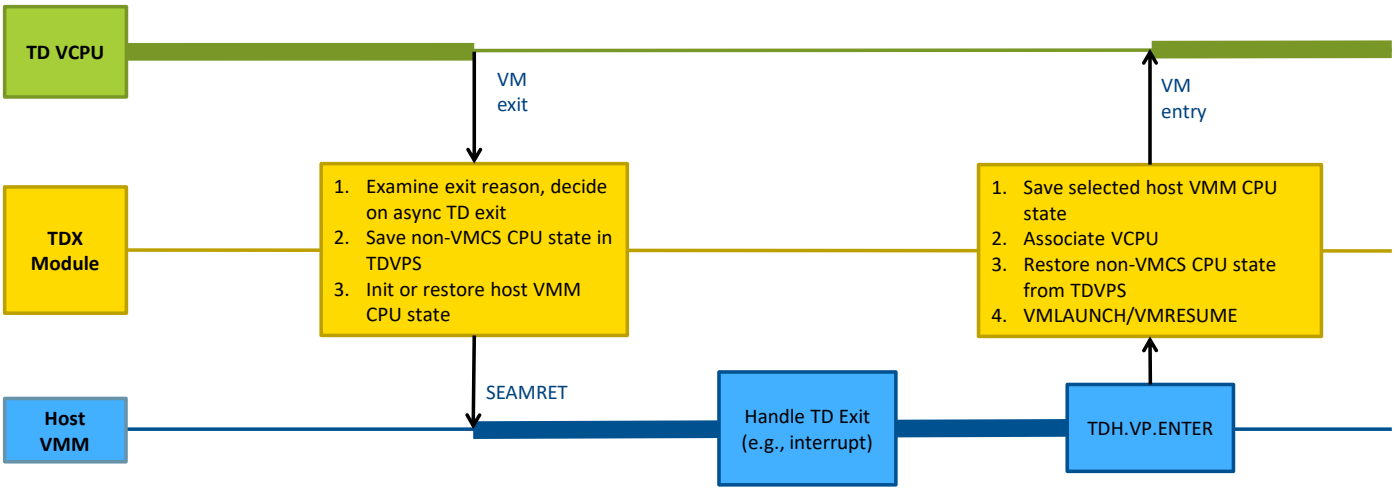

Figure 9.2: Example of Asynchronous TD Exit and TD Resumption

9.1.2. 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 definition in 22.2.40 and TDG.VP.VMCALL in 22.3.10.

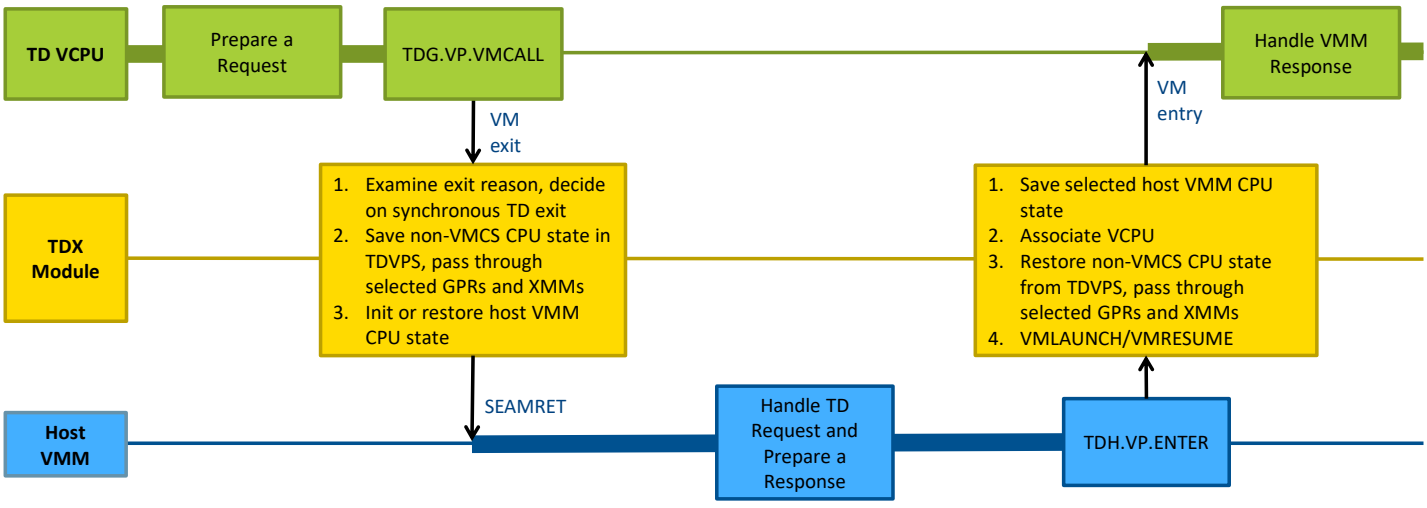

Figure 9.3: Example of Synchronous TD Exit and TD Resumption

9.1.3. VCPU Activity State Machine

The VCPU activity state machine, controlled by TDVPS.STATE as shown in Table 9.1 below and shown in Figure 9.4 below, helps ensure the following:

- A VCPU can be entered only when its logical TDVPS control structure, composed of TDVPR and TDVPX pages, is available in memory and has been initialized.
- 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 9.1: TDVPS.STATE Definition

State Name		Description
VCPU_UNINITIALIZED		VCPU has not been initialized yet by TDH.VP.INIT.
VCPU_READY		This is a super-state with 2 sub-states. In this super-state, VCPU can be entered.
	VCPU_READY_ASYNC	Last TD exit was due to an asynchronous event (non-TDG.VP.VMCALL). VCPU state has been fully saved on TD exit and will be restored on the next TD entry.
	VCPU_READY_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.
VCPU_ACTIVE		VCPU is active (TDX non-root mode) on some LP.
VCPU_DISABLED		VCPU is disabled.

TD Entry and TD Exit transitions normally toggle between the VCPU_READY super-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.

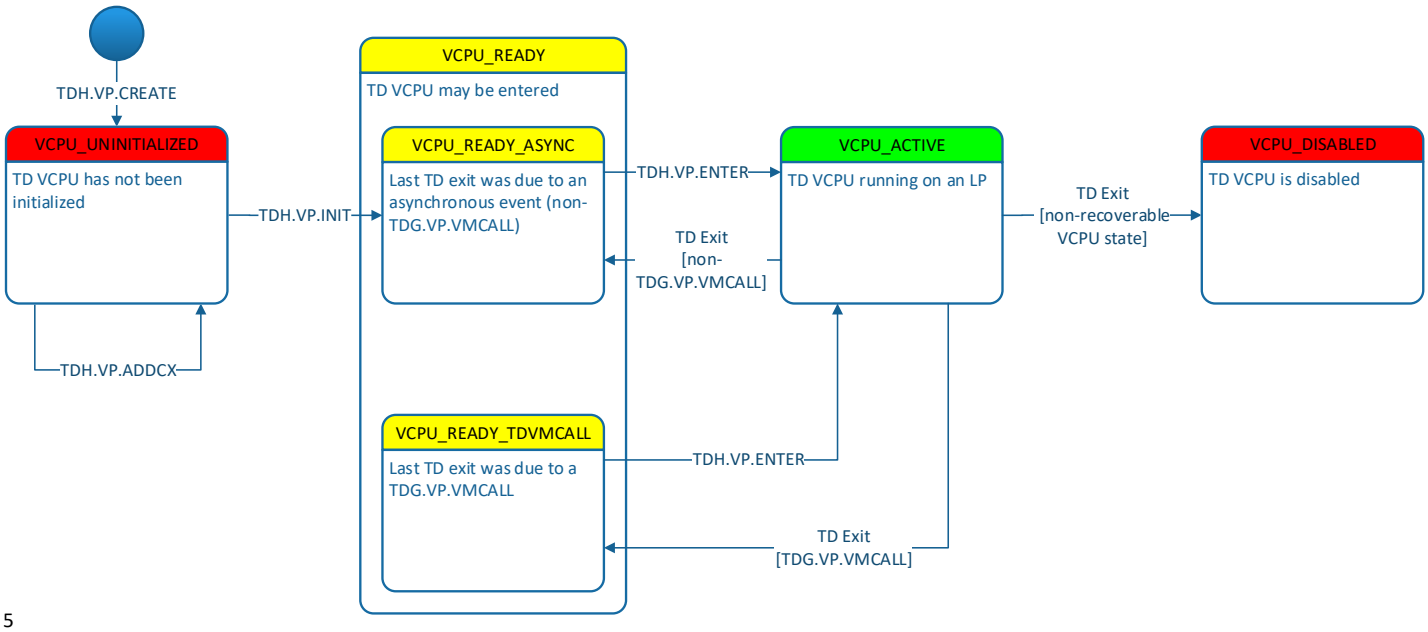

Figure 9.4: VCPU Activity State Machine

9.2. 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.

9.2.1. TD ASID Components

Table 9.2 below shows a high-level view of the components of the TD ASID. The exact structure is micro-architectural.

Table 9.2: 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

9.2.2. 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.

An alternative method the host VMM may use is to do TLB tracking similar to how it's done for Secure EPT, i.e., execute TDH.MEM.TRACK and a round of IPI. Contrary to Secure EPT, this is not enforced by the TDX module.

9.3. VCPU-to-LP Association

9.3.1. 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.

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

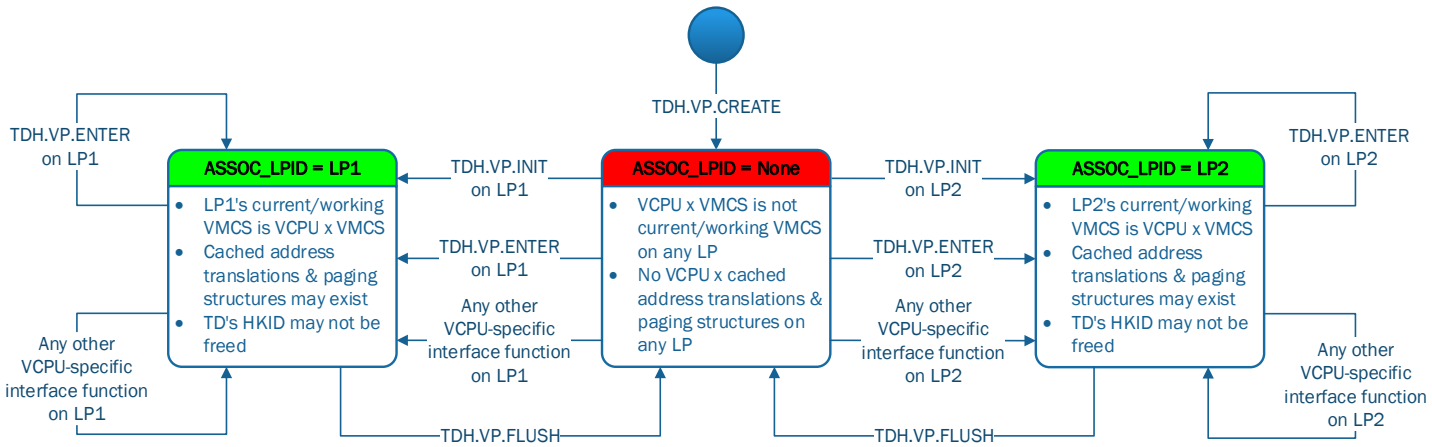

Figure 9.5: VCPU Association State Machine

The following Intel TDX module mechanisms are designed to help ensure 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 22.2.25.
- 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 4.4.

9.3.3. 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.

10.CPU Virtualization (Non-Root Mode Operation)

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

10.1. Initial State

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

10.1.1. 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 10.1.
- The IA32_EFER MSR is initialized to support the CPU modes described in 10.1.
- The initial values of some GPRs provide some basic information to the guest TD as described in 10.1.2 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 21.2.

10.1.2. Initial State of Guest TD GPRs

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

Table 10.1: 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)

10.1.3. 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 10.6.1 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 10.6.2 for details.

10.1.4. 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).

10.1.5. 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.

10.2. 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 10.1 below shows a typical flow used by guest software.

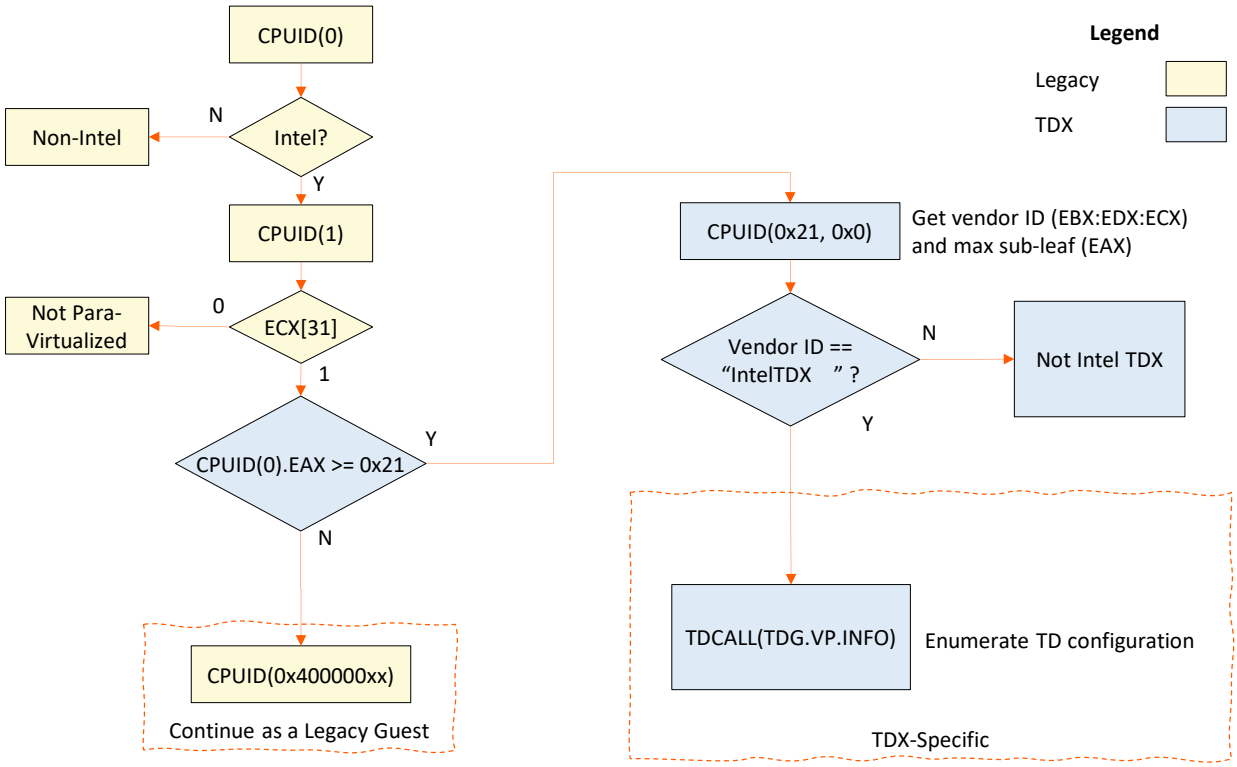

Figure 10.1: Typical Run-Time Environment Enumeration by a Guest TD

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 10.2: 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.

10.3. 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 10.3: 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.

10.4. 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 10.10.

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

10.4.1. 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

10.4.2. Instructions that Cause a #VE Unconditionally

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

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

- PCONFIG (see 10.15)

10.4.4. 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).

10.5. 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

10.5.1. 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 10.4 below.

10.5.2. Extended State Isolation

The Intel TDX module helps ensure 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.

10.5.3. 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 10.4: 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.

10.6. CR Handling

10.6.1. 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

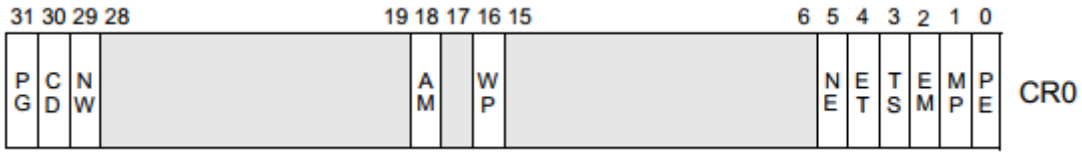

Figure 10.2: CR0

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.

10.6.2. 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 10.14 below.

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.

10.7. MSR Handling

10.7.1. 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 18.1.

10.8. CPUID Virtualization

10.8.1. 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 10.5: 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.	
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 is described in 22.2.17, and its input TD_PARAMS structure is described in 20.3. Configuration is further classified as follows:

Table 10.6: CPUID Configuration by the TD_PARAMS Input of TDH.MNG.INIT

TD_PARAMS Section	Description
CPUID_CONFIG	Bit fields configurable directly based on a configuration table
XFAM	Bit fields configurable based on the guest TD's XFAM XFAM control of extended features virtualization is described in 10.5.
ATTRIBUTES	Bit fields configurable based on the guest TD's ATTRIBUTES
Other	Bits fields configurable based on some other field of TD_PARAMS

A detailed list of CPUID virtualization is provided in 18.2. 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.INFO, described in 13.1.3.3.

10.8.2. Unconditional #VE for all CPUID Leaves and Sub-Leaves

The guest TD may use the TDG.VP.CPUIDVE.SET to toggle on or off the unconditional injection of #VE on all CPUID leaves and sub-leaves. 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. TDG.VP.CPUIDVE.SET is described in 22.3.7.

10.9. 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

10.9.1. 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.

10.9.2. 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.

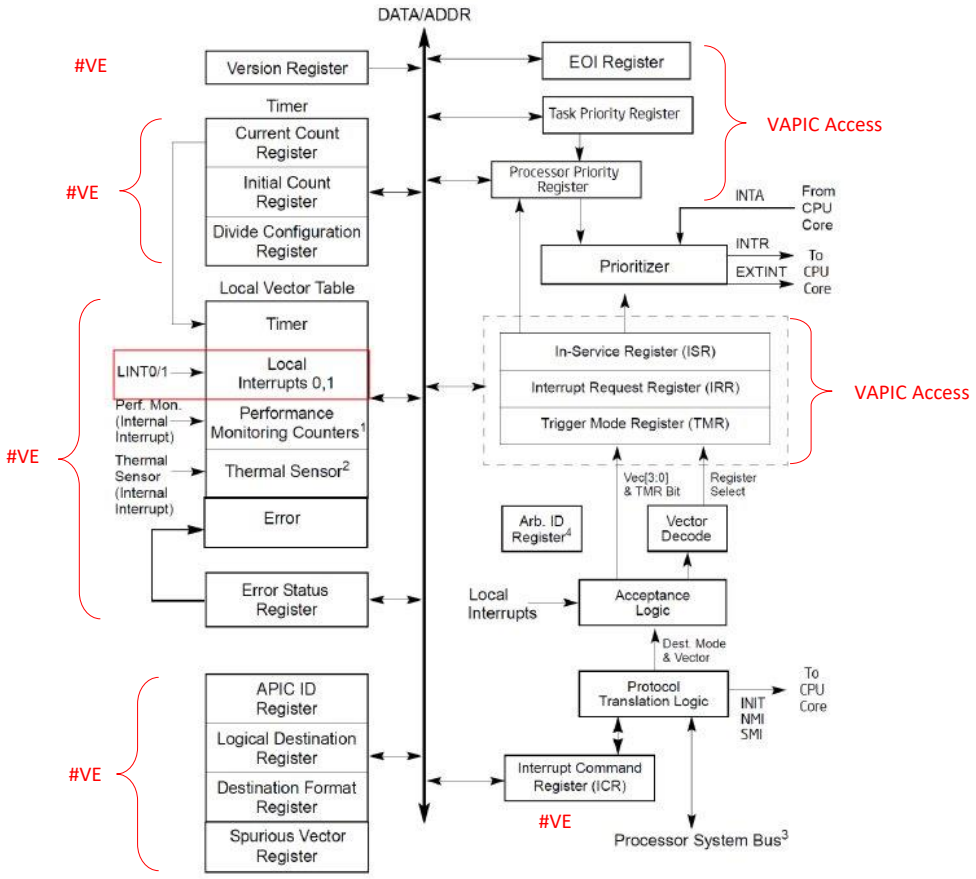

Figure 10.3: Virtual APIC Access by Guest TD

Table 10.7: 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])

10.9.3. Implicit APIC Write #VE

The following guest operations result in an APIC write VM exit to the TDX module. The VM exit is trap-like, i.e., it happens after the instruction has been executed:

- WRMSR of IA32_X2APIC_SELF_IPI with EAX[7:4] set to 0, i.e., an interrupt vector value smaller than 16.
- Executing SENDUIPI to send a user-level interrupt.

In all such cases, the TDX module injects a #VE exception back to the guest TD, with the exit reason indicating an APIC write and bits 11:0 of the exit qualification set to the page offset of the write access.

10.9.4. 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.

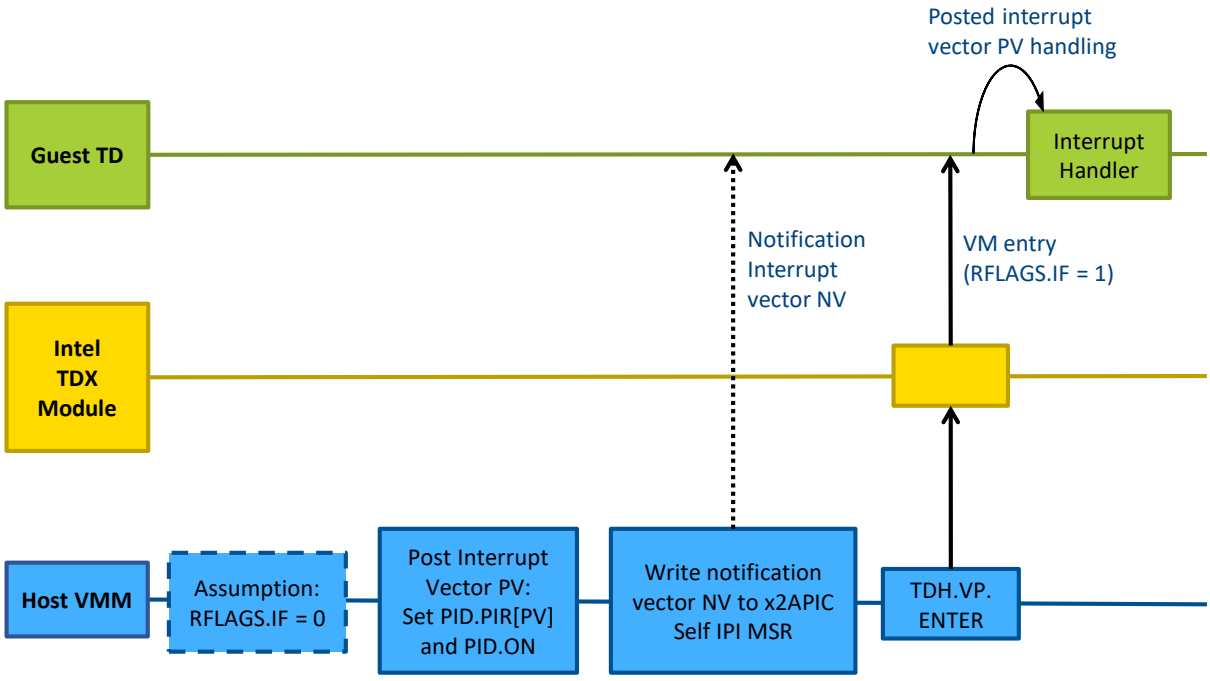

Figure 10.4: Typical Sequence for Posted Interrupt Injection to the Current LP

10.9.5. 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 21.2.3.

10.9.6. 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 10.9.4 above. This is an untrusted operation; thus, the TD needs to track its completion.

10.9.7. 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.

10.10. 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 (e.g., #GP(0)) is not applicable. 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.

10.10.1. 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 22.3.7). This is a simple way to help ensure the availability and privacy of this area.

Table 10.8: 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.

Section	Field	Offset (Bytes)	Size (Bytes)	Description
	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 10.10.2 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 22.3.7). TDG.VP.VEINFO.GET checks that VE_INFO.VALID is 0xFFFFFFFF. After reading the information, it sets VE_INFO.VALID to 0.

10.10.2. #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 SEPT_PENDING (an exception to this is described in 10.11.1 below). If ATTRIBUTES.SEPT_VE_DISABLE is 0, the Intel TDX module sets the Secure EPT entry's Suppress VE bit (63) to 0 if the entry's state is SEPT_PENDING. It sets that bit to 1 for all other entry states.

10.10.3. #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 10.4 above), a disallowed MSR access (see 10.7 above), or CPUID virtualization (see 10.8 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 16.3 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.

10.11. 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 8.

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

10.11.1. 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.

10.11.2. 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 SEPT_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 10.10 above.

10.12. 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.

10.12.1. 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.

10.12.2. 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.

10.12.3. 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.

Bus Lock VM Exit Reason (74)

If "bus-lock detection" is enabled, then if the processor detects that one or more bus locks were caused by the instruction that was executed, then the processor generates a bus lock VM exit (exit reason 74). This VM exit is trap-like, i.e., it is delivered following the execution of that instruction that caused it. The Intel TDX module then completes a TD exit with the exit information provided in the VM exit.

Bus Lock Detected Bit (26) in VM Exit Reason

If delivery of bus lock VM exit was pre-empted by a higher priority VM exit (e.g., EPT Misconfiguration, EPT Violation, etc.), then the processor sets a "bus lock detected" notification bit (bit 26) in the exit reason. The Intel TDX module reflects this bit to the host VMM on TD exit.

10.12.4. 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.

10.13. 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

10.13.1. 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.

10.14. Supervisor Protection Keys (PKS)

By design, guest TD usage of Supervisor Protection Keys (PKS) is controlled by the ATTRIBUTES.PKS bit (see 20.3.1). 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.

10.15. 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.

10.15.1. 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).

10.15.2. 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.

10.16. Virtualization of Machine Check Capabilities and Controls

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 MCE 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.

10.17. Other Changes in TDX Non-Root Mode

10.17.1. 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.

10.17.2. 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.

11.Measurement and Attestation

11.1. 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.

11.1.1. 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, TDVPR and TDVPX) 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.

11.1.2. 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.

11.2. 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 is detailed in 20.7, and TDG.MR.REPORT is detailed in 22.3.3.

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 11.3 below.

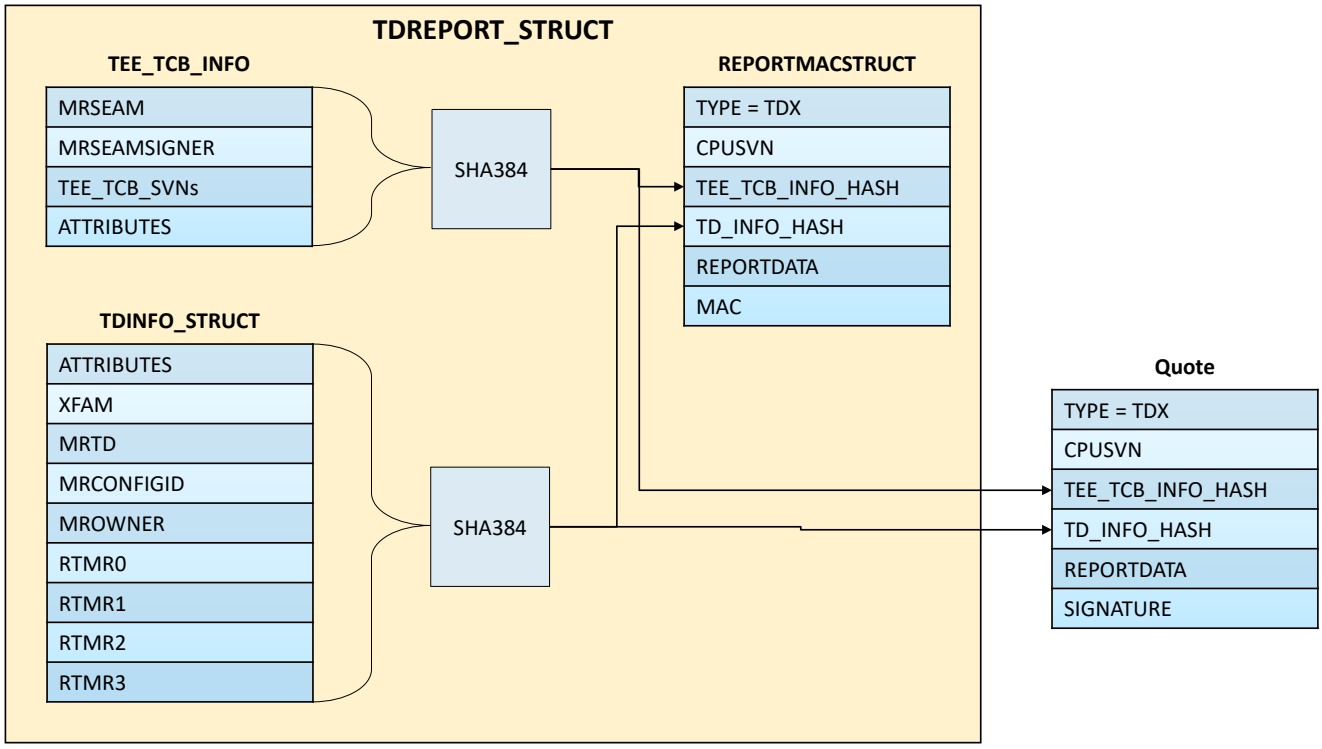

Figure 11.1: TDX Measurement Reporting

11.3. 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.

11.3.1. 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.

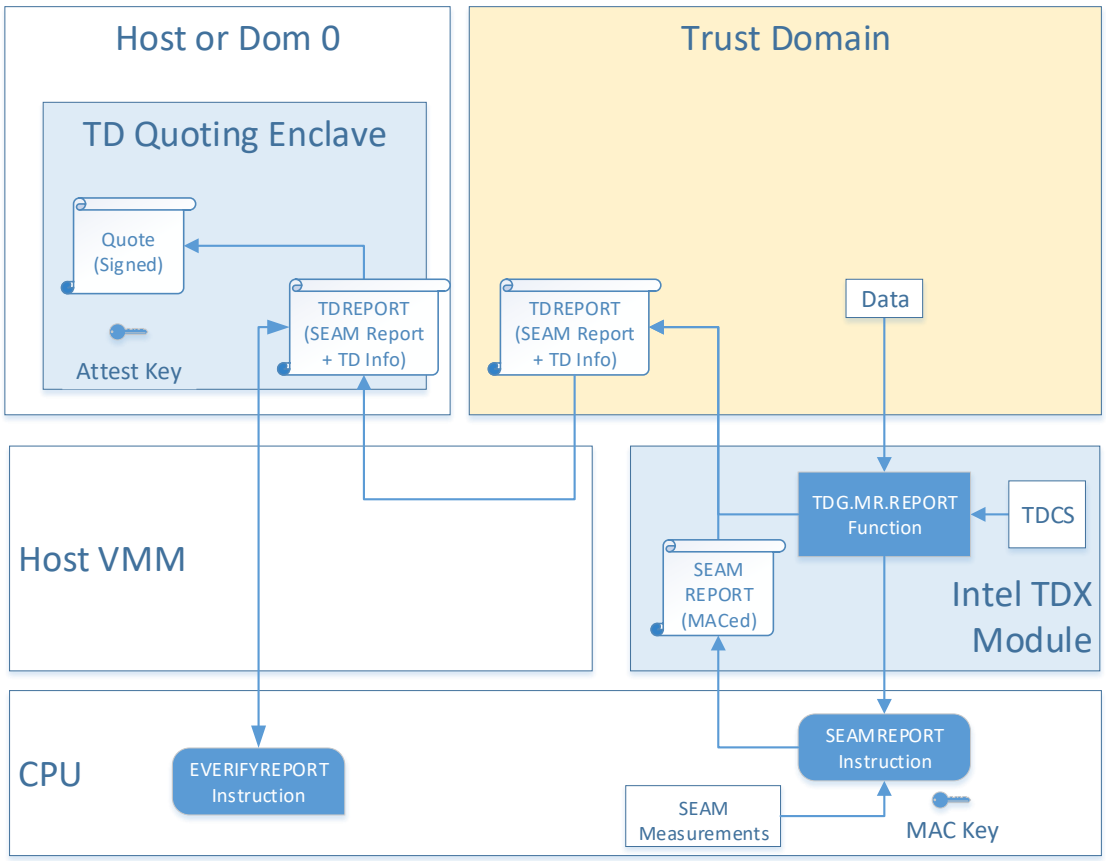

Figure 11.2: High-Level View of the Intel SGX-Based TD Attestation

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.

11.4. 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.

11.5. 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.

12.I/O Support

This chapter specifies the Intel TDX I/O model.

12.1. 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.

12.2. 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.

12.3. 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 10.10. 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.

12.4. 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.

12.5. 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.

12.6. 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.

13.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.

13.1. Overview

13.1.1. Initialization and Configuration Flow

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

Table 13.1: 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 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.INFO	The host VMM calls 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.

13.1.2. 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.

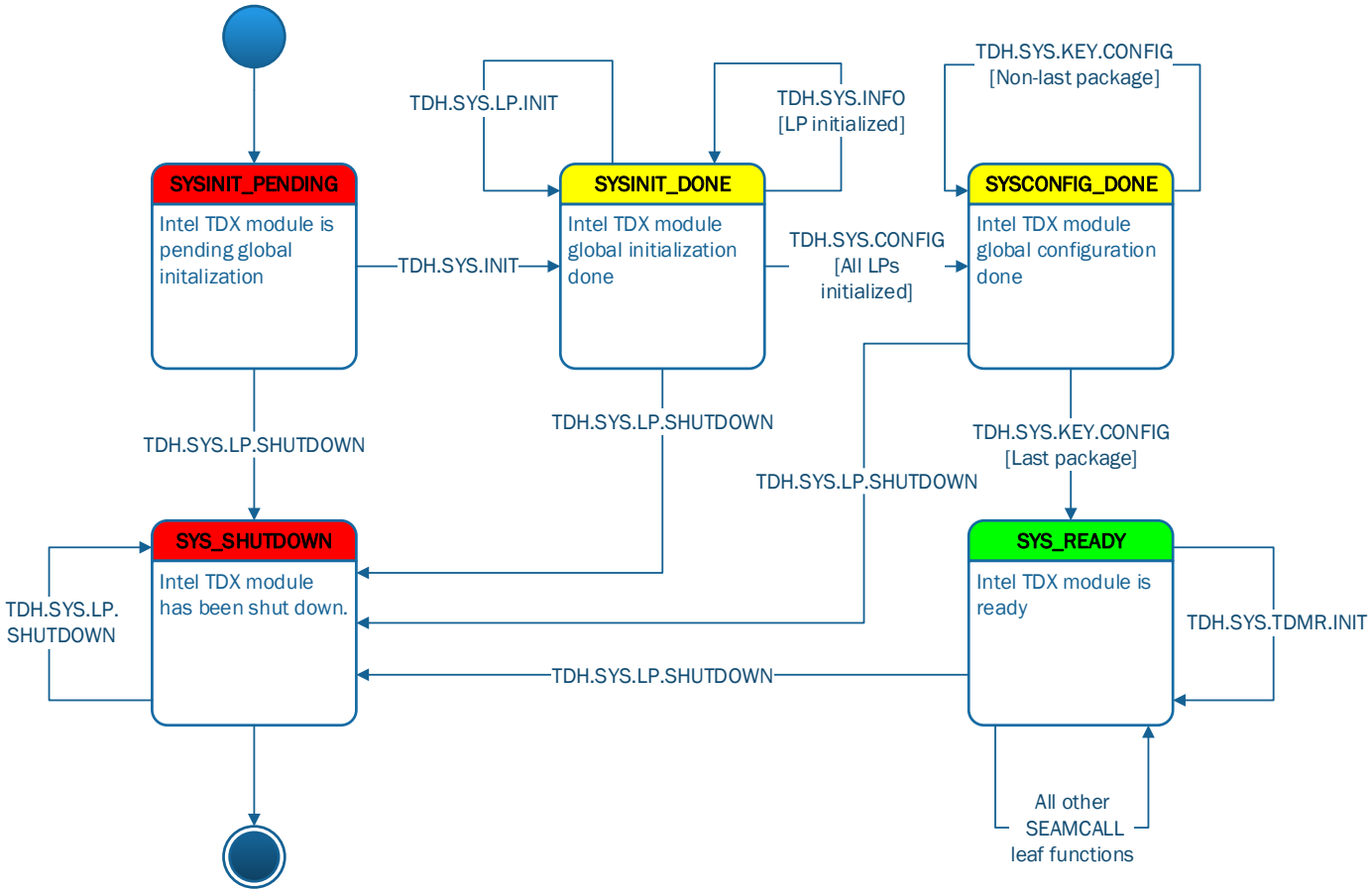

Figure 13.1: Intel TDX Module Lifecycle State Machine

Table 13.2: Intel TDX Module Lifecycle States

State Name	Description	Allowed SEAMCALL Leaf Functions
SYSINIT_PENDING	TDH.SYS.INIT has not been called yet.	TDH.SYS.INIT TDH.SYS.SHTDOWNLP
SYSINIT_DONE	TDH.SYS.INIT has completed successfully. TDH.SYS.LP.INIT must be called on each LP.	TDH.SYS.LP.INIT TDH.SYS.INFO (if current LP has been initialized) TDH.SYS.CONFIG (if all LPs have been initialized) TDH.SYS.SHTDOWNLP
SYSCONFIG_DONE	TDH.SYS.CONFIG has completed successfully. TDH.MNG.KEY.CONFIG must be called on each package.	TDH.SYS.KEY.CONFIG TDH.SYS.INFO TDH.SYS.SHTDOWNLP
SYS_READY	The Intel TDX module is ready for use.	Any
SYS_SHUTDOWN	Shutdown operation has been initiated by TDH.SYS.LP.SHUTDOWN. No new host-side interface functions can be called.	TDH.SYS.SHTDOWNLP (once per LP)

13.1.3. Platform Compatibility and Configuration Checking

13.1.3.1. *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.

13.1.3.2. *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.

13.1.3.3. *CPUID Sampling, Checks and Enumeration*

Note: CPUID virtualization is described in 10.8.

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.INFO function may be called by the host VMM to enumerate the directly configurable and allowable CPUID fields, using the TDSYSINFO_STRUCT described in 20.8.1.

13.1.4. 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.

13.1.4.1. *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.

13.1.4.2. *TDMRs and PAMT Arrays Configuration*

TDMRs and PAMTs are described in 7.1. This section provides an overview of their configuration and their relationships to CMRs.

13.1.4.2.1. Background: Reserved Areas within TDMRs

As described in 7.1, 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.

13.1.4.2.2. Background: Three PAMT Areas

As described in 7.1, 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.

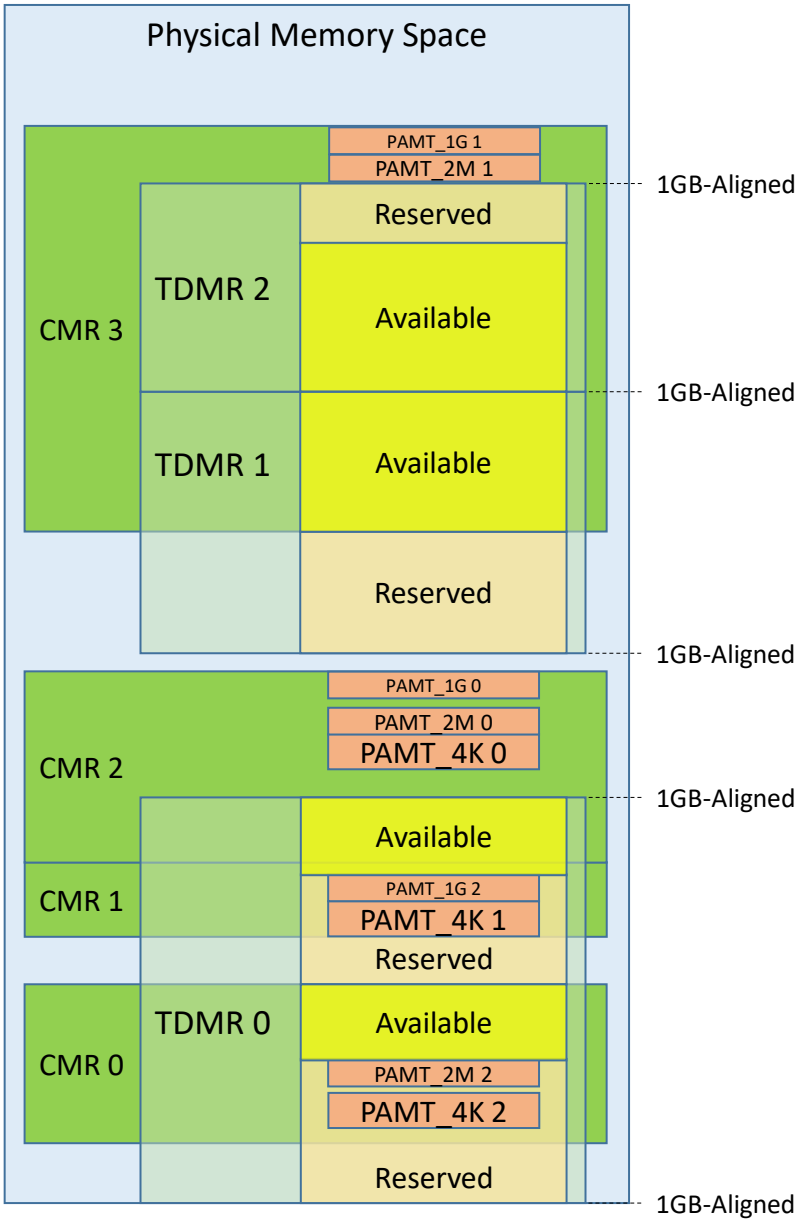

Figure 13.2: Example of Convertible Memory Ranges (CMRs) vs. Trust Domain Memory Regions (TDMRs)

13.1.4.2.3. Configuration Rules

In addition to the rules described in 7.1, 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.INFO.
- 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 TDMR 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.

13.2. Intel TDX Module Initialization Interface

13.2.1. 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 22.2.33.

13.2.2. 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 22.2.33.

13.2.3. Enumeration: TDH.SYS.INFO

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

- Intel TDX module capabilities are enumerated in the returned TDSYSINFO_STRUCT (see 20.8.1).
- Convertible Memory Ranges (CMRs), as previously set by BIOS and checked by MCHECK, are enumerated in the returned CMR_INFO table (see 20.8.3).

For a detailed description of TDH.SYS.INFO, see 22.2.32.

13.2.4. 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. The table format (TDMR_INFO) is described in 20.8.4. Refer to 7.1 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 TDH.SYS.CONFIG, see 22.2.31.

13.2.5. 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 22.2.32.

13.3. 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 22.2.37.

13.4. Intel TDX Module Shutdown

13.4.1. 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.

13.4.2. 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.

14.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

14.1. On-TD Debug

Intel SDM, Vol. 3, 17 Debug, Branch Profile, TSC and Intel Resource Director Technology (Intel RDT) Features

14.1.1. 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 10.5)
- PT (if allowed by the TD's XFAM, see 10.5)
- BTS

However, the TDX architecture does not allow guest TDs to use BTM.

14.1.2. Generic Debug Handling

14.1.2.1. *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.

14.1.2.2. *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 14.4 below).
  - Enable BTM by setting bits 7:6 to 0x1 (see details in 14.1.3 below).
- Uncore PMI is virtualized as disabled; bit 13 is read as 0 (see 14.4 below).

14.1.3. Debug Feature-Specific Handling

The following table discusses how specific debug features are handled.

Table 14.1: 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 14.1.2 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 14.2 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 10.12.4.
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 10.5.
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 10.5.
Non-Architectural LBRs	• IA32_DEBUGCTL bit 0 (LBR)	Guest TD attempt to set IA32_DEBUGCTL[0] is ignored by the CPU.

14.2. On-TD Performance Monitoring

Intel SDM, Vol. 3, 18 Performance Monitoring

14.2.1. 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 20.3.1).

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 the RDPMC instruction and the Perfmon MSRs (see 14.2.2 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 14.4 below for details.

See also 14.1 above.

The Intel TDX module is designed to support performance monitoring as implemented on the GLC core:

- 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 14.2.2 below)

14.2.2. Performance Monitoring MSRs

Perfmon uses the following MSRs:

Table 14.2: 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			

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 10.7.

14.2.3. 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 10.9.4), or as virtual NMI using the NMI injection interface (see 10.9.6).

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 14.4 below.

14.3. 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.

The applicable Intel TDX functions are listed in Table 14.3 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 14.3: Off-TD Debug Interface

Intel TDX Function	ATTRIBUTES.DEBUG = 0	ATTRIBUTES.DEBUG = 1	References
TDH.MNG.RD/ TDH.MNG.WR	N/A	Access secret and non-secret TD-scope state in TDR and TDCS.	21.1, 22.2.22, 22.2.23
TDH.MEM.SEPT.RD	Read Secure EPT entry	Read Secure EPT entry	22.2.12

Intel TDX Function	ATTRIBUTES.DEBUG = 0	ATTRIBUTES.DEBUG = 1	References
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).	21.2
TDH.MEM.WR/ TDH.MEM.RD	N/A	Access TD-private memory.	22.2.25, 22.2.25
TDH.PHYMEM.PAGE.RDMD	Read page metadata (PAMT information)	Read page metadata (PAMT information).	22.2.28

14.3.1. 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.

14.3.2. 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.

14.4. 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.

15. Memory Integrity Protection and Machine Check Handling

15.1. Overview

The Intel TDX module's memory integrity protection and machine check handling are designed to answer address the following security objectives:

- 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 by the Intel TDX module.
- To help improve resistance to brute force attacks, software must not be able to **repeatedly** cause memory integrity violations during Intel TDX module or guest TD operation. When an integrity violation is detected, the affected guest TD and the key corresponding to its affected HKID must become unusable for normal operation of the TD – i.e., the TD may only be torn down.
- Any software except guest TD or TDX module 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.

As a **best effort**, the TDX module is designed to enable limiting the impact of memory integrity violations in a guest TD context to that guest TD, i.e., requiring only that guest TD to be torn down. However, there are cases where memory integrity violations result in an unbreakable shutdown of the LP.

15.2. TDX Memory Integrity Protection Background

15.2.1. Cryptographic Integrity (Ci) vs. Logical Integrity (Li), MAC and TD Owner

TDX architecture aims to provide resiliency against confidentiality and integrity attacks by software. Towards this goal, the TDX architecture helps enforce the enabling of memory integrity for all private HKIDs. It supports two memory integrity modes that can be configured on the platform:

Cryptographic Integrity (Ci) Memory content is encrypted and protected by a MAC and a TD Owner bit.

Logical Integrity (Li) Memory content is encrypted and protected by a TD Owner bit.

In both Ci and Li modes, the memory controllers store a 1-bit **TD Owner** metadata each cache. The TD Owner bit is set to 1 for writes with a private HKID and is cleared to 0 for writes with a shared HKID. The TD Owner bit is covered by ECC.

When Ci mode is enabled, the CPU's memory controllers compute a 28-bit integrity check value (**MAC**) for the data (cache line) during writes, and store the MAC with the memory as meta-data. The MAC is calculated over the components described in the table below. The MAC is covered by ECC.

Table 15.1: Components for MAC Calculation (Ci Mode)

Component	Description
Ciphertext Data	512 bits of data being written to memory.
Encryption Tweak	128-bit encryption tweak, generated by encrypting the physical address with the 128-bit per-HKID ephemeral AES-XTS tweak key. The tweak key is generated on key configuration (TDH.SYS.KEY.CONFIG and TDH.MNG.KEY.CONFIG).
TD Owner Bit	Indicates that the data was written using a private HKID.
MAC Key	128-bit MAC key, generated by hardware on platform initialization, when BIOS configures the IA32_TME_ACTIVATE MSR.

15.2.2. MAC and TD Owner Update on Memory Writes

The MAC and the TD Owner bit are updated on memory writes by the memory controller per the following criteria:

- If memory write is for a private HKID, the TD Owner bit is set, and integrity information (MAC) is computed and stored as meta-data along with ciphertext in memory.
- Else (write is for a shared HKID), the TD Owner bit is clear, and based on the key configuration, integrity information (MAC) may be stored along with ciphertext in memory.

The state diagram below shows the TD Owner bit state changes due to memory state changes.

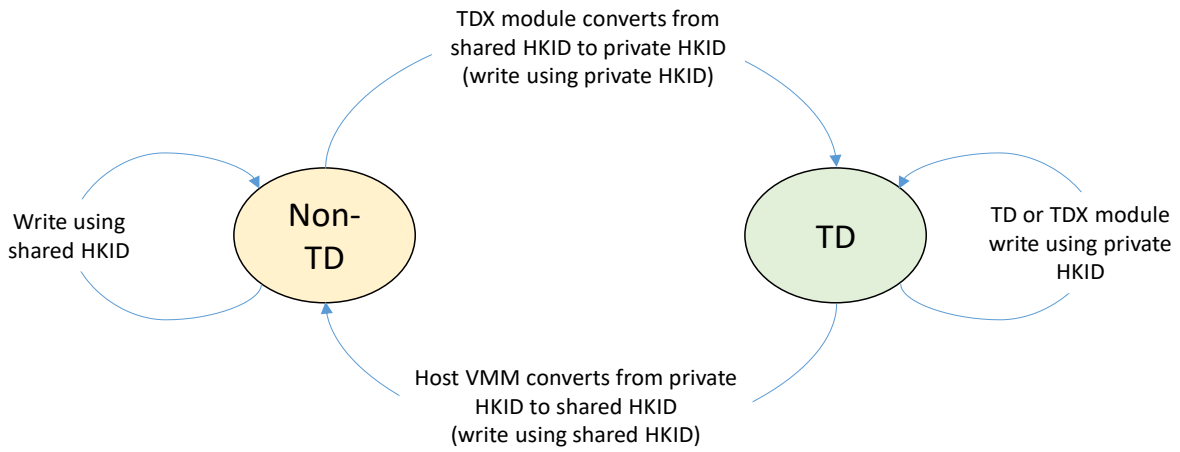

Figure 15.1: TD Owner Bit Setting on Write

15.2.3. No Checks on Memory Writes

The TD Owner bit is not checked on memory writes. It is the responsibility of the host VMM to prevent writing to memory that has been assigned as TD private memory. Failing to do so will result in a memory corruption; such corruption will be detected when the guest TD or the TDX module attempts to read that memory, as described above.

15.2.4. Checks on Memory Reads

Checks on memory reads depend on whether Cryptographic Integrity (Ci) is enabled on the platform, or Logical Integrity (Li) is used. This is shown in the tables below.

Any reads of TD private data (TD Owner is 1) done outside SEAM mode return all-0. This is intended to prevent the host VMM from testing malicious ciphertext for a MAC collision, since the VMM will deterministically see zeroed data in the cache for speculative accesses, which on subsequent non-speculative accesses will cause a machine check event. No poison indication is returned.

Table 15.2: Checks on Memory Reads in Ci Mode

HKID Type	Integrity Enabled for HKID	TD Owner Bit	Integrity Check	Returned Data	Poison
Private	Yes	0	Pass	0	Poison
		0	Fail	0	Poison
		1	Pass	Decrypted data	None
		1	Fail	0	Poison
Shared	Yes	0	Pass	Decrypted data	None
		0	Fail	0	Poison
		1	Pass	0	Poison
		1	Fail	0	Poison
Shared	No	0	N/A	Decrypted data	None
		1	N/A	0	None

Table 15.3: Checks on Memory Reads in Li Mode

HKID Type	Integrity Enabled for HKID	TD Owner Bit	Integrity Check	Returned Data	Poison
Private	No	0	N/A	0	Poison
		1	N/A	Decrypted data	None
Shared	No	0	N/A	Decrypted data	None
		1	N/A	0	None

Memory integrity errors that result in poison generation are logged by the memory controller as **UCNA** (uncorrected no-action required) UCR errors which are signaled via CMCI (if CMCI is enabled) or CSMI (if enabled).

On a subsequent consumption (read) of the poisoned data by software, there are two possible scenarios:

Machine Check: In most cases, the core determines that the execution can continue, and it treats poison with fault-like exception semantics signaled as an **MCE** (Machine Check Exception) or **MSMI** (Machine-check System Management Interrupt).

Handling of machine check events (MCE or MSMI) when executing in a guest TD (in SEAM non-root mode) and in the Intel TDX module (in SEAM root mode) is described in the following sections.

Unbreakable Shutdown: In some cases, the core determines that execution cannot continue (e.g., long µCode flows), and it goes into an unbreakable shutdown.

An unbreakable shutdown that happens while running in SEAM mode, either in a guest TD or in the TDX module, globally marks TDX as disabled – all subsequent SEAMCALL invocations on any logical processor of the platform lead to a VMFailInvalid error.

15.3. Machine Check Architecture (MCA) Background

Intel SDM, Vol. 3, 15 Machine-Check Architecture

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.

15.3.1. Uncorrected Machine Check Error

The processor signals the detection of an **uncorrected machine-check error** by generating a **machine-check exception (MCE)**, which is a fault-like exception. An MCA enhancement supports software recovery from certain uncorrected **recoverable** machine check errors. Poisoned cache line consumption by the guest TD is considered such an error. The machine-check exception handler is expected to be implemented in the VMM.

15.3.2. Corrected Machine Check Interrupt (CMCI)

Intel SDM, Vol. 3, 15.5 Corrected Machine Check Error Interrupt

Processors on which TDX will be supported can also 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 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.

15.3.3. Machine Check System Management Interrupt (MSMI)

MSMI is part of the Enhanced Machine Check Architecture, Gen. 2 (EMCA2). With EMCA2 enabled, each machine check bank can be configured to assert SMI instead of MCE or CMCI. This is intended to allow the SMM handler to correct the error when possible. For details, see [Error Reporting through EMCA2].

15.3.4. Local Machine Check Event (LMCE)

Intel SDM, Vol. 3, 15.3.1.5 Enabling Local Machine Check

When system software has enabled LMCE, then hardware will determine if a particular error can be delivered only to a single logical processor, instead of being broadcast to all logical processors. This is the recommended configuration for TDX.

15.4. Recommended MCA Platform Configuration for TDX

The following platform MCA configuration is recommended for TDX:

- LMCE should be enabled, so that machine check events that happen in the scope of a certain logical processor are delivered only to that logical processor.
- EMCA2 should be disabled on core MC banks (IFU and DCU), so that the host VMM can handle memory integrity errors by tearing down a single TD, instead of shutting down the whole platform.

The following sections provide additional details.

15.5. Handling Machine Check Events during Guest TD Operation

15.5.1. Machine Check Events Delivered as an #MC Exception (Recommended)

If EMCA2 is not enabled on the core MC banks (IFU and DCU), the machine check event is delivered as an #MC exception. With LMCE enabled, the MCE is delivered only to the logical processor that consumed the poisoned cache line.

The Intel TDX module configures the MCE events when they occur in a TD guest to cause a VM exit to the Intel TDX module. This includes the following cases:

- MCE during guest TD operation
- MCE during a successful VM entry to a guest TD
- MCE during a failed VM exit, where normally execution would remain in the guest TD

The Intel TDX module implements this as follows:

- The Intel TDX module enforces guest TD CR4.MCE to 1.
- The Intel TDX module sets bit 18 (MC) of the TD VMCS Exception Bitmap to 1.

On VM exit, if the exit reason is Exception or NMI (0), the Intel TDX module reads the TD VMCS' VM-exit interruption information to determine if the VM exit was caused by a #MC (18). If so, the Intel TDX module puts the TD in a FATAL state, preventing further TD entries. The TDX module then completes the TD exit flow. The TDH.VP.ENTER outputs indicate the status as TDX_NON_RECOVERABLE_TD_FATAL and provides the exit reason, exit qualification and exit interruption information.

Note: The TDX module does not analyze the MCE to determine its source – whether it's a memory integrity violation or some other event.

Based on the TDH.VM.ENTER outputs (exit reason etc.), the host VMM is expected to understand that a Machine Check event happened, and that the TD should be torn down.

The host VMM can reclaim memory assigned to TDs in a FATAL state using the normal TD teardown flow (TDH.VP.FLUSH, TDH.MNG.VPFLUSHDONE, TDH.PHYMEM.CACHE.WB, TDH.MNG.KEY.FREEID, TDH.PHYMEM.PAGE.RECLAIM).

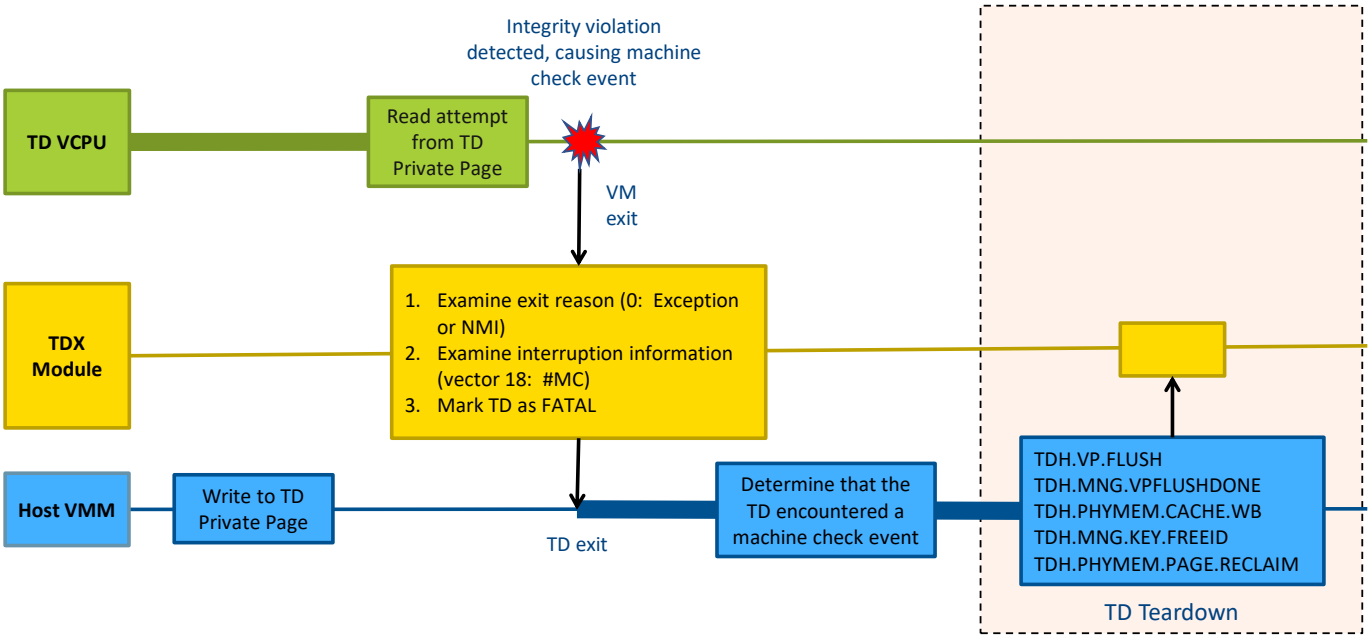

Figure 15.2: Example of Handling an MCE in a TD Context

15.5.2. Machine Check Events Delivered as an MSMI (Not Recommended)

If EMCA2 is enabled on the core MC banks (IFU and DCU), the machine check event is delivered as an MSMI. With LMCE enabled, the MSMI is delivered only to the logical processor that consumed the poisoned cache line.

Contrary to non-TDX operation, an SMI that occurs in a TD guest does not immediately invoke the SMM handler. Instead, an SMI causes a VM exit to the Intel TDX module and remains pending.

On VM exit, if the exit reason is Other SMI (6), the Intel TDX module reads the TD VMCS' exit qualification bit 0 to determine if the VM exit was caused by a Machine Check that was mutated into an SMI. If so, the Intel TDX module puts the TD in a FATAL state, preventing further TD entries. The TDX module then completes the TD exit flow. The TDH.VP.ENTER outputs indicate the status as TDX_NON_RECOVERABLE_TD_FATAL and provides the exit reason and exit qualification.

Note: The TDX module does not analyze the MCE to determine its source – whether it's a memory integrity violation or some other event.

Once TD exit has completed and the CPU is no longer in SEAM mode, the pending SMI event is taken and the platform's SMM handler is invoked. On RSM, the SMM handler injects an #MC to the host VMM.

The host VMM needs to understand that the reported machine check event happened during TD execution and may be handled by tearing down the TD. However, this may not be simple to implement. Thus, **EMCA2 enabling on the core MC banks is not recommended for TDX**.

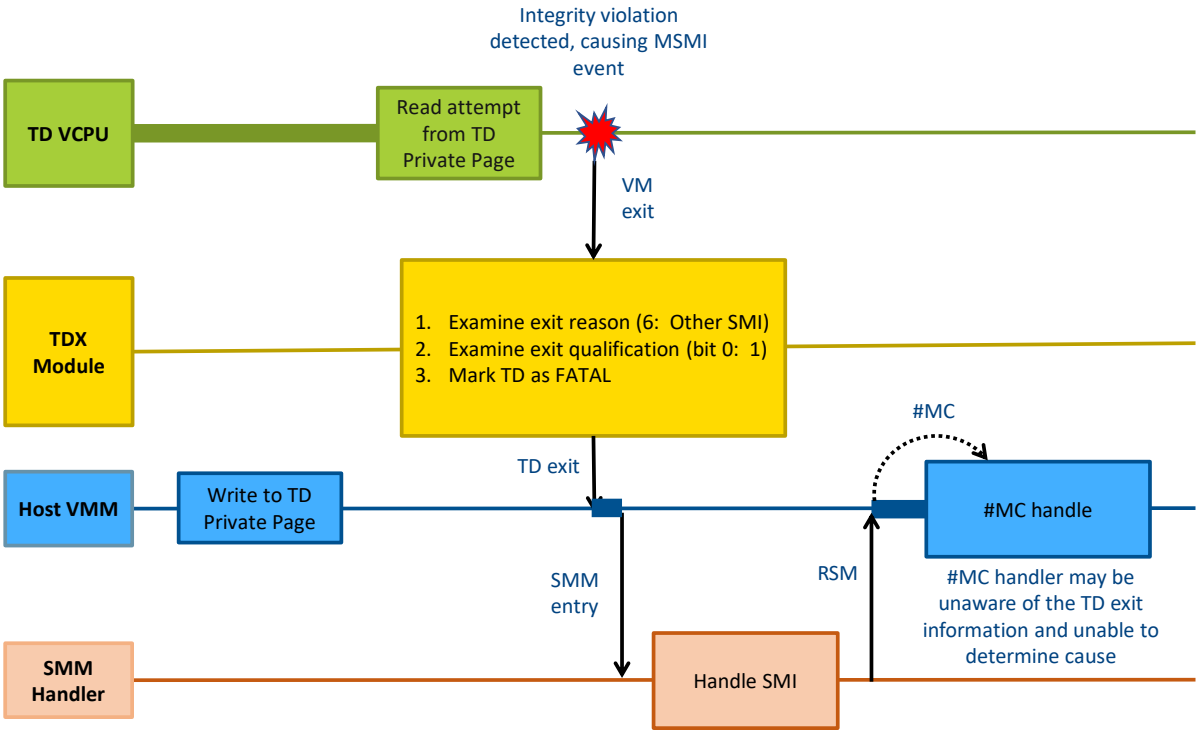

Figure 15.3: Example of Handling an MSMI in a TD Context

15.5.3. LMCE Disabled (Not Recommended)

If LMCE is disabled, then an MCE or MSMI is broadcast to all logical processors on the platform. Any TD that happens to be running will be put in a FATAL state.

Note: The TDX module does not check the MCE details. Any MCE that causes a VM exit from a guest TD is considered fatal to that TD.

15.5.4. Machine Check Events Delivered as a CMCI

CMCI is treated as a normal interrupt, causing an asynchronous TD exit; there's no special handling.

On VM exit, if the exit reason is Exception or NMI (0), the Intel TDX module reads the TD VMCS' VM-exit interruption information to determine if the VM exit was caused by a #MC (18). If not, the Intel TDX module completes the TD exit flow. The TDH.VP.ENTER outputs indicate the status as TDX_SUCCESS and provides the exit reason, exit qualification and exit interruption information.

Based on the TDH.VM.ENTER outputs, the host VMM is expected to process the CMCI interrupt.

15.6. Handling MCE during Intel TDX Module Operation

Any machine check event that occurs during Intel TDX module operation (in SEAM root mode) forces an unbreakable shutdown on a current LP. Shutdown also globally marks TDX as disabled – all subsequent SEAMCALL invocations on any logical processor of the platform lead to a VMFailInvalid error.

16.Side Channel Attack Mitigation Mechanisms

16.1. 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)

16.2. 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 ensure that by applying CPU mechanisms to isolate the branch predictions of each guest TD from branch predication done outside its execution.

16.3. Single-Step and Zero-Step Attacks Mitigation Mechanisms

16.3.1. 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.

16.3.2. 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.

16.3.3. 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.

17.General Aspects of the Intel TDX Interface Functions

17.1. Concurrency Restrictions and Enforcement

17.1.1. 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 17.1: 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 17.2: 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.

17.1.2. 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.

17.1.3. 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.

17.2. 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)

17.2.1. 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

17.2.1.1. *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 17.3: 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 • TDVPX
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.

17.2.1.2. *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.

17.2.1.3. *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)**.

17.2.1.4. *Memory Type*

17.2.1.4.1. Memory Type for Private and Opaque Accesses

The memory type for **private** and **opaque** access semantics, which use a private HKID, is WB.

17.2.1.4.2. 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.

17.2.1.5. *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.

17.2.1.6. *Summary Table*

Table 17.4: 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

17.3. 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

17.3.1. 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.

17.3.2. 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.

17.3.2.1. *Least Detailed Level: Success/Warning/Error*

At this simplest level, software can differentiate among three cases:

Table 17.5: Intel TDX Interface Functions Completion Status in RAX at the Least Detailed Level

RAX	Meaning	Description
0	Success	Function completed successfully
> 0 (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
< 0 (0x80000000_00000000 – 0xFFFFFFFF_FFFFFFFF)	Error	Function aborted due to some error

17.3.2.2. *Medium Detailed Level: Class and Recoverability*

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.

17.3.2.3. *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 TDVPX pages or due to the VCPU already being initialized.

Table 17.6: 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.
61:48	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

Bits	Name	Description
		• CMR index • TDMR index

Refer to 19.1 for a list of function completion codes.

17.3.3. Other CPU State Convention

All Intel TDX functions except TDH.VP.ENTER are designed to preserve the CPU state not explicitly defined as output.

TDH.VP.ENTER is a special case. In addition to explicit output operands discussed in 17.3.4 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 22.2.31).

17.3.4. Transition Cases: TD Entry and Exit

17.3.4.1. *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 22.3.10.

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.VMCAL) 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 22.3.10.

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 22.2.31).

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.

17.3.4.2. *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 21.2). 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.

17.4. Metadata Access Interface

17.4.1. 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.
- 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 and write of TDR, TDCS and TDVPS control structures by the host VMM using the SEAMCALL functions TDH.MNG.RD, TDH.MNG.WR, TDH.VP.RD and TDH.VP.WR
- Read and write of TDCS control structure by the guest TD using the following TDCALL functions TDG.VM.RD and TDG.VM.WR

17.4.2. Metadata Fields and Elements

Metadata fields are identified by **field identifiers (MD_FIELD_ID)**. A field identifier contains a **FIELD_CODE** and other information – for a detailed description see 20.9 and MD_FIELD_ID values are defined in tables provided in Ch. 21. 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 17.1 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 17.1: Example of a 48 Byte TDCS.MRCONFIGID Field Composed of Six 8 Byte Elements

A detailed definition of a field identifier is provided in 20.9.

17.4.3. Arrays of Metadata Fields

Metadata fields can be organized in arrays. Figure 17.2 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 17.2: Example of an Array of 4 Single-Element Fields

Figure 17.3 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 17.3: Example of an Array of Four 48 Byte TDCS.RTMR Fields, Each Composed of 6 Elements

17.5. 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 16.3) is activated due to a suspected attack.

SECTION 3: INTEL TDX APPLICATION BINARY INTERFACE (ABI) REFERENCE

This section serves as a detailed reference for the Intel TDX module ABI. This includes:

- CPU virtualization tables
- Constants
- Data types
- Variables
- Control structures
- functions

The reference contains software-visible details. It also contains some internal details at a level required for understanding the architecture.

18.ABI Reference: CPU Virtualization Tables

18.1. MSR Virtualization

Table 18.2 below describes how the Intel TDX module virtualizes MSRs to guest TDs. The table uses a notation that is described in Table 18.1 below.

Table 18.1: MSR Virtualization Notation Definition

Text	Virtualization
Native	Direct read or write from/to CPU

Inject_GP(condition)	TDX Module injects a #GP(0) if condition is true, else reads from CPU or write to CPU:
	`if (condition)`
	`  #GP(0)`
	`else`
	`  Native`

Inject_GP_or_VE(condition)	TDX Module injects a #GP(0) if condition is true, else it injects a #VE:
	`if (condition)`
	`  #GP(0)`
	`else`
	`  #VE`

For MSRs that are not listed in the table, the Intel TDX module injects a #VE on both RDMSR and WRMSR by the guest TD.

Note: The table below provides a high-level overview of MSR virtualization. Implementation details may differ.

Table 18.2: MSR Virtualization

MSR Index Range (Hex)				MSR Virtualization	
First (Hex)	**Last (Hex)**	**Size (Hex)**	**MSR Architectural Name**	**On RDMSR**	**On WRMSR**
0x0010	0x0010	0x1	IA32_TIME_STAMP_COUNTER	Native	#VE
0x0048	0x0048	0x1	IA32_SPEC_CTRL	Native	Native
0x0049	0x0049	0x1	IA32_PRED_CMD	Native	Native
0x0087	0x0087	0x1	IA32_MKTME_PARTITIONING	Inject_GP_or_VE (~virt. CPUID(7,0).EDX[18])	Inject_GP_or_VE (~virt. CPUID(7,0).EDX[18])
0x008C	0x008F	0x4	IA32_SGXLEPUBKEYHASHx	#GP(0)	#GP(0)
0x0098	0x0098	0x1	MSR_WBINVDP	#GP(0)	#GP(0)
0x0099	0x0099	0x1	MSR_WBNOINVDP	#GP(0)	#GP(0)
0x009A	0x009A	0x1	MSR_INTR_PENDING	#GP(0)	#GP(0)
0x009B	0x009B	0x1	IA32_SMM_MONITOR_CTL	#GP(0)	#GP(0)
0x009E	0x009E	0x1	IA32_SMBASE	#GP(0)	#GP(0)
0x00C1	0x00C8	0x8	IA32_PMCx	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x00E1	0x00E1	0x1	IA32_UMWAIT_CONTROL	Inject_GP(~virt. CPUID(7,0).ECX[5])	Inject_GP(~virt. CPUID(7,0).ECX[5])
0x010A	0x010A	0x1	IA32_ARCH_CAPABILITIES	Get the value read on TDX module init, clear the following bits: Bit 7 (TSX_CTRL) Bit 10 (MISC_PACKAGE_CTLS) Bit 11 (ENERGY_FILTERING_CTL) Bits 63:12 (reserved)	Native
0x010B	0x010B	0x1	IA32_FLUSH_CMD	Native	Native
0x0122	0x0122	0x1	IA32_TSX_CTRL	#GP(0)	#GP(0)
0x0174	0x0174	0x1	IA32_SYSENTER_CS	Native	Native
0x0175	0x0175	0x1	IA32_SYSENTER_ESP	Native	Native
0x0176	0x0176	0x1	IA32_SYSENTER_EIP	Native	Native
0x0186	0x018D	0x8	IA32_PERFEVTSELx	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)

MSR Index Range (Hex)				MSR Virtualization	
First (Hex)	Last (Hex)	Size (Hex)	MSR Architectural Name	On RDMSR	On WRMSR
0x01A0	0x01A0	0x1	IA32_MISC_ENABLE	if ~PERFMON RDMSR current value Indicate Perfmon and PEBS are unavailable: Bit 7 = 0 Bit 12 = 1 else Native	#VE
0x01A6	0x01A7	0x2	MSR_OFFCORE_RSPx	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x01C4	0x01C4	0x1	IA32_XFD	Inject_GP(~(virt. CPUID(0xD,0x1).EAX[4]))	Inject_GP(~(virt. CPUID(0xD,0x1).EAX[4]))
0x01C5	0x01C5	0x1	IA32_XFD_ERR	Inject_GP(~(virt. CPUID(0xD,0x1).EAX[4]))	Inject_GP(~(virt. CPUID(0xD,0x1).EAX[4]))
0x01D9	0x01D9	0x1	IA32_DEBUGCTL	Clear ENABLE_UNCORE_PMI (bit 13)	#GP if illegal, #VE if value is not supported for TD
0x01F8	0x01F8	0x1	IA32_PLATFORM_DCA_CAP	Inject_GP_or_VE(~virt. CPUID(0x1).ECX[18])	Inject_GP_or_VE(~virt. CPUID(0x1).ECX[18])
0x01F9	0x01F9	0x1	IA32_CPU_DCA_CAP	Inject_GP_or_VE(~virt. CPUID(0x1).ECX[18])	Inject_GP_or_VE(~virt. CPUID(0x1).ECX[18])
0x01FA	0x01FA	0x1	IA32_DCA_0_CAP	Inject_GP_or_VE(~virt. CPUID(0x1).ECX[18])	Inject_GP_or_VE(~virt. CPUID(0x1).ECX[18])
0x0276	0x0276	0x1	MSR_SLAM_ENABLE	#GP(0)	#GP(0)
0x0277	0x0277	0x1	IA32_PAT	Native	Native
0x0309	0x030C	0x4	IA32_FIXED_CTRx	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0329	0x0329	0x1	IA32_PERF_METRICS	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0345	0x0345	0x1	IA32_PERF_CAPABILITIES	if ~PERFMON return 0 else if ~XFAM[8] clear bit 16 else Native	Inject_GP(~PERFMON)
0x038D	0x038D	0x1	IA32_FIXED_CTR_CTRL	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x038E	0x038E	0x1	IA32_PERF_GLOBAL_STATUS	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x038F	0x038F	0x1	IA32_PERF_GLOBAL_CTRL	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0390	0x0390	0x1	IA32_PERF_GLOBAL_STATUS_RESET	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0391	0x0391	0x1	IA32_PERF_GLOBAL_STATUS_SET	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0392	0x0392	0x1	IA32_PERF_GLOBAL_INUSE	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x03F1	0x03F1	0x1	IA32_PEBS_ENABLE	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x03F2	0x03F2	0x1	MSR_PEBS_DATA_CFG	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x03F6	0x03F6	0x1	MSR_PEBS_LD_LAT	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x03F7	0x03F7	0x1	MSR_PEBS_FRONTEND	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0480	0x0480	0x1	IA32_VMX_BASIC	#GP(0)	#GP(0)
0x0481	0x0481	0x1	IA32_VMX_PINBASED_CTLS	#GP(0)	#GP(0)
0x0482	0x0482	0x1	IA32_VMX_PROCBASED_CTLS	#GP(0)	#GP(0)
0x0483	0x0483	0x1	IA32_VMX_EXIT_CTLS	#GP(0)	#GP(0)
0x0484	0x0484	0x1	IA32_VMX_ENTRY_CTLS	#GP(0)	#GP(0)
0x0485	0x0485	0x1	IA32_VMX_MISC	#GP(0)	#GP(0)
0x0486	0x0486	0x1	IA32_VMX_CR0_FIXED0	#GP(0)	#GP(0)
0x0487	0x0487	0x1	IA32_VMX_CR0_FIXED1	#GP(0)	#GP(0)
0x0488	0x0488	0x1	IA32_VMX_CR4_FIXED0	#GP(0)	#GP(0)
0x0489	0x0489	0x1	IA32_VMX_CR4_FIXED1	#GP(0)	#GP(0)
0x048A	0x048A	0x1	IA32_VMX_VMCS_ENUM	#GP(0)	#GP(0)
0x048B	0x048B	0x1	IA32_VMX_PROCBASED_CTLS2	#GP(0)	#GP(0)
0x048C	0x048C	0x1	IA32_VMX_EPT_VPID_CAP	#GP(0)	#GP(0)
0x048D	0x048D	0x1	IA32_VMX_TRUE_PINBASED_CTLS	#GP(0)	#GP(0)
0x048E	0x048E	0x1	IA32_VMX_TRUE_PROCBASED_CTLS	#GP(0)	#GP(0)
0x048F	0x048F	0x1	IA32_VMX_TRUE_EXIT_CTLS	#GP(0)	#GP(0)
0x0490	0x0490	0x1	IA32_VMX_TRUE_ENTRY_CTLS	#GP(0)	#GP(0)
0x0491	0x0491	0x1	IA32_VMX_VMFUNC	#GP(0)	#GP(0)
0x0492	0x0492	0x1	IA32_VMX_PROCBASED_CTLS3	#GP(0)	#GP(0)
0x04C1	0x04C8	0x8	IA32_A_PMCx	Inject_GP(~PERFMON)	Inject_GP(~PERFMON)
0x0500	0x0500	0x1	IA32_SGX_SVN_STATUS	#GP(0)	#GP(0)
0x0560	0x0560	0x1	IA32_RTIT_OUTPUT_BASE	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0561	0x0561	0x1	IA32_RTIT_OUTPUT_MASK_PTRS	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0570	0x0570	0x1	IA32_RTIT_CTL	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0571	0x0571	0x1	IA32_RTIT_STATUS	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0572	0x0572	0x1	IA32_RTIT_CR3_MATCH	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0580	0x0580	0x1	IA32_RTIT_ADDR0_A	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])

MSR Index Range (Hex)				MSR Virtualization	
First (Hex)	Last (Hex)	Size (Hex)	MSR Architectural Name	On RDMSR	On WRMSR
0x0581	0x0581	0x1	IA32_RTIT_ADDR0_B	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0582	0x0582	0x1	IA32_RTIT_ADDR1_A	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0583	0x0583	0x1	IA32_RTIT_ADDR1_B	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0584	0x0584	0x1	IA32_RTIT_ADDR2_A	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0585	0x0585	0x1	IA32_RTIT_ADDR2_B	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0586	0x0586	0x1	IA32_RTIT_ADDR3_A	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0587	0x0587	0x1	IA32_RTIT_ADDR3_B	Inject_GP(~XFAM[8])	Inject_GP(~XFAM[8])
0x0600	0x0600	0x1	IA32_DS_AREA	Native	Native
0x06A0	0x06A0	0x1	IA32_U_CET	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06A2	0x06A2	0x1	IA32_S_CET	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06A4	0x06A4	0x1	IA32_PL0_SSP	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06A5	0x06A5	0x1	IA32_PL1_SSP	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06A6	0x06A6	0x1	IA32_PL2_SSP	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06A7	0x06A7	0x1	IA32_PL3_SSP	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06A8	0x06A8	0x1	IA32_INTERRUPT_SSP_TABLE_ADDR	Inject_GP(~(XFAM[11] | XFAM[12]))	Inject_GP(~(XFAM[11] | XFAM[12]))
0x06E1	0x06E1	0x1	IA32_PKRS	Inject_GP(~PKS)	Inject_GP(~PKS)
0x0800	0x0801	0x2	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x0804	0x0807	0x4	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x0808	0x0808	0x1	IA32_X2APIC_TPR	Native	Native
0x0809	0x0809	0x1	Reserved for xAPIC MSRs	Native	Native
0x080A	0x080A	0x1	IA32_X2APIC_PPR	Native	Native
0x080B	0x080B	0x1	IA32_X2APIC_EOI	Native	Native
0x080C	0x080C	0x1	Reserved for xAPIC MSRs	Native	Native
0x080E	0x080E	0x1	Reserved for xAPIC MSRs	Native	Native
0x0810	0x0817	0x8	IA32_X2APIC_ISRx	Native	Native
0x0818	0x081F	0x8	IA32_X2APIC_TMRx	Native	Native
0x0820	0x0827	0x8	IA32_X2APIC_IRRx	Native	Native
0x0829	0x082E	0x6	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x0831	0x0831	0x1	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x083F	0x083F	0x1	IA32_X2APIC_SELF_IPI	Native	Native
0x0840	0x087F	0x40	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x0880	0x08BF	0x40	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x08C0	0x08FF	0x40	Reserved for xAPIC MSRs	#GP(0)	#GP(0)
0x0981	0x0981	0x1	IA32_TME_CAPABILITY	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])
0x0982	0x0982	0x1	IA32_TME_ACTIVATE	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])
0x0983	0x0983	0x1	IA32_TME_EXCLUDE_MASK	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])
0x0984	0x0984	0x1	IA32_TME_EXCLUDE_BASE	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])	Inject_GP_or_VE (~virt. CPUID(7,0).ECX[13])
0x0985	0x0985	0x1	IA32_UINT_RR	Inject_GP(~XFAM[14])	Inject_GP(~XFAM[14])
0x0986	0x0986	0x1	IA32_UINT_HANDLER	Inject_GP(~XFAM[14])	Inject_GP(~XFAM[14])
0x0987	0x0987	0x1	IA32_UINT_STACKADJUST	Inject_GP(~XFAM[14])	Inject_GP(~XFAM[14])
0x0988	0x0988	0x1	IA32_UINT_MISC	Inject_GP(~XFAM[14])	Inject_GP(~XFAM[14])
0x0989	0x0989	0x1	IA32_UINT_PD	Inject_GP(~XFAM[14])	Inject_GP(~XFAM[14])
0x098A	0x098A	0x1	IA32_UINT_TT	Inject_GP(~XFAM[14])	Inject_GP(~XFAM[14])
0x0C80	0x0C80	0x1	IA32_DEBUG_INTERFACE	Native	#VE
0x0D90	0x0D90	0x1	IA32_BNDCFGS	#GP(0)	#GP(0)
0x0D93	0x0D93	0x1	IA32_PASID	#GP(0)	#GP(0)

0x0DA0	0x0DA0	0x1	IA32_XSS	Native	if illegal or does not match XFAM
					  #GP(0)
					else
					  Write to CPU

0x1200	0x12FF	0x100	IA32_LBR_INFO	Inject_GP(~XFAM[15])	Inject_GP(~XFAM[15])
0x14CE	0x14CE	0x1	IA32_LBR_CTL	Inject_GP(~XFAM[15])	Inject_GP(~XFAM[15])
0x14CF	0x14CF	0x1	IA32_LBR_DEPTH	Inject_GP(~XFAM[15])	Inject_GP(~XFAM[15])
0x1500	0x15FF	0x100	IA32_LBR_FROM_IP	Inject_GP(~XFAM[15])	Inject_GP(~XFAM[15])
0x1600	0x16FF	0x100	IA32_LBR_TO_IP	Inject_GP(~XFAM[15])	Inject_GP(~XFAM[15])
0xC0000080	0xC0000080	0x1	IA32_EFER	Native	#VE
0xC0000081	0xC0000081	0x1	IA32_STAR	Native	Native
0xC0000082	0xC0000082	0x1	IA32_LSTAR	Native	Native
0xC0000084	0xC0000084	0x1	IA32_FMASK	Native	Native
0xC0000100	0xC0000100	0x1	IA32_FSBASE	Native	Native
0xC0000101	0xC0000101	0x1	IA32_GSBASE	Native	Native
0xC0000102	0xC0000102	0x1	IA32_KERNEL_GS_BASE	Native	Native
0xC0000103	0xC0000103	0x1	IA32_TSC_AUX	Native	Native

18.2. CPUID Virtualization

Table 18.4 below describes how the Intel TDX module virtualizes CPUID to guest TDs. Note the following:

- The "Configuration by TDH.MNG.INIT" column details which section of the TD_PARAMS structure is used for configuring how each CPUID bit field is virtualized.
- The "Virtualization" column uses a notation defined in Table 18.3 below.
- If the guest TD executes CPUID with a valid leaf / sub-leaf number combination that is not listed in the table, the Intel TDX module injects a #VE.
- The host VMM should always consult the list of CPUID leaves and sub-leaves configured by TD_PARAMS.CPUID_CONFIG, as enumerated by TDH.SYS.INFO, described in 13.1.3.3.

Table 18.3: CPUID Virtualization Notation Definition

CPUID Bit Field Virtualization	Meaning	Virtualization Details
As Configured	Virtual bit field value reflects the host VMM configuration.	
As Configured (if Native)	If the native bit field value returned by executing CPUID is 0, then the virtual bit field value is o. Else, the virtual bit field value reflects the host VMM configuration.	
Calculated	Bit field is calculated by the Intel TDX module.	Calculation method
Fixed	The virtual bit field value is fixed.	Bit field value
Native	The virtual bit field value reflects the native value returned by executing CPUID.	

Note: The table below provides a high-level overview of CPUID virtualization. Implementation details may differ.

Table 18.4: CPUID Virtualization Overview

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
				Leaf 0x0				
EAX	31	0	32	MaxIndex	N/A		Fixed	0x21
EBX	31	0	32	Genu	N/A		Native	
ECX	31	0	32	ntel	N/A		Native	
EDX	31	0	32	inel	N/A		Native	
				Leaf 0x1				
EAX	3	0	4	Stepping ID	N/A		Calculated	Min. of all packages
EAX	7	4	4	Model ID	N/A		Native	
EAX	11	8	4	Family ID	N/A		Native	
EAX	13	12	2	Processor Type	N/A		Native	
EAX	15	14	2	Reserved_15_14	N/A		Fixed	0x0
EAX	19	16	4	Extended Model ID	N/A		Native	
EAX	27	20	8	Extended Family ID	N/A		Native	
EAX	31	28	4	Reserved_31_28	N/A		Fixed	0x0
EBX	7	0	8	Brand Index	N/A		Native	
EBX	15	8	8	CLFLUSH Line Size	N/A		Fixed	0x8
EBX	23	16	8	Maximum Addressable IDs	CPUID_CONFIG		As Configured	
EBX	31	24	8	Initial APIC ID	N/A		Calculated	TDVPS.VCPU_INDEX[7:0]
ECX	0	0	1	SSE3	N/A		Native	
ECX	1	1	1	PCLMULQDQ	N/A		Native	
ECX	2	2	1	DTES64	N/A		Native	
ECX	3	3	1	MONITOR	N/A		Fixed	0x0
ECX	4	4	1	DS-CPL	N/A		Native	
ECX	5	5	1	VMX	N/A		Fixed	0x0
ECX	6	6	1	SMX	N/A		Fixed	0x0
ECX	7	7	1	EST	CPUID_CONFIG		As Configured (if Native)	
ECX	8	8	1	TM2	CPUID_CONFIG		As Configured (if Native)	
ECX	9	9	1	SSSE3	N/A		Native	

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
ECX	10	10	1	CNXT-ID	N/A		Native	
ECX	11	11	1	SDBG	N/A		Native	
ECX	12	12	1	FMA	XFAM	XFAM[2]	As Configured (if Native)	
ECX	13	13	1	CMPXCHG16B	N/A		Fixed	0x1
ECX	14	14	1	xTPR Update Control	CPUID_CONFIG		As Configured (if Native)	
ECX	15	15	1	PDCM	N/A		Fixed	0x1
ECX	16	16	1	Reserved_16	N/A		Fixed	0x0
ECX	17	17	1	PCID	N/A		Native	
ECX	18	18	1	DCA	CPUID_CONFIG		As Configured (if Native)	
ECX	19	19	1	SSE4_1	N/A		Native	
ECX	20	20	1	SSE4_2	N/A		Native	
ECX	21	21	1	x2APIC	N/A		Fixed	0x1
ECX	22	22	1	MOVBE	N/A		Native	
ECX	23	23	1	POPCNT	N/A		Native	
ECX	24	24	1	TSC-Deadline	N/A		Native	
ECX	25	25	1	AESNI	N/A		Fixed	0x1
ECX	26	26	1	XSAVE	N/A		Fixed	0x1
ECX	27	27	1	OSXSAVE	N/A		Calculated	CR4.OSXSAVE
ECX	28	28	1	AVX	XFAM	XFAM[2]	As Configured (if Native)	
ECX	29	29	1	F16C	XFAM	XFAM[2]	As Configured (if Native)	
ECX	30	30	1	RDRAND	N/A		Fixed	0x1
ECX	31	31	1	Reserved_31	N/A		Fixed	0x1
EDX	0	0	1	FPU	N/A		Native	
EDX	1	1	1	VME	N/A		Native	
EDX	2	2	1	DE	N/A		Native	
EDX	3	3	1	PSE	N/A		Native	
EDX	4	4	1	TSC	N/A		Native	
EDX	5	5	1	MSR	N/A		Fixed	0x1
EDX	6	6	1	PAE	N/A		Fixed	0x1
EDX	7	7	1	MCE	N/A		Fixed	0x1
EDX	8	8	1	CX8	N/A		Native	
EDX	9	9	1	APIC	N/A		Fixed	0x1
EDX	10	10	1	Reserved_10	N/A		Fixed	0x0
EDX	11	11	1	SEP	N/A		Native	
EDX	12	12	1	MTRR	N/A		Fixed	0x1
EDX	13	13	1	PGE	N/A		Native	
EDX	14	14	1	MCA	N/A		Fixed	0x1
EDX	15	15	1	CMOV	N/A		Native	
EDX	16	16	1	PAT	N/A		Native	
EDX	17	17	1	PSE-36	N/A		Native	
EDX	18	18	1	PSN	N/A		Native	
EDX	19	19	1	CLFSH	N/A		Fixed	0x1
EDX	20	20	1	Reserved_20	N/A		Fixed	0x0
EDX	21	21	1	DS	N/A		Fixed	0x1
EDX	22	22	1	ACPI	CPUID_CONFIG		As Configured (if Native)	
EDX	23	23	1	MMX	N/A		Native	
EDX	24	24	1	FXSR	N/A		Native	
EDX	25	25	1	SSE	N/A		Native	
EDX	26	26	1	SSE2	N/A		Native	
EDX	27	27	1	SS	N/A		Native	
EDX	28	28	1	HTT	CPUID_CONFIG		As Configured (if Native)	
EDX	29	29	1	TM	CPUID_CONFIG		As Configured (if Native)	
EDX	30	30	1	Reserved_30	N/A		Fixed	0x0
EDX	31	31	1	PBE	CPUID_CONFIG		As Configured (if Native)	
Leaf 0x3								
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0x4 / Sub-Leaf 0x0								
EAX	4	0	5	Type	CPUID_CONFIG		As Configured	
EAX	7	5	3	Level	CPUID_CONFIG		As Configured	
EAX	8	8	1	Self Initializing	CPUID_CONFIG		As Configured	
EAX	9	9	1	Fully Associative	CPUID_CONFIG		As Configured	
EAX	13	10	4	Reserved	CPUID_CONFIG		As Configured	
EAX	25	14	12	Addressable IDs Sharing this Cache	CPUID_CONFIG		As Configured	
EAX	31	26	6	Addressable IDs for Cores in Package	CPUID_CONFIG		As Configured	
EBX	11	0	12	L	N/A		Native	

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
EBX	21	12	10	P	CPUID_CONFIG		As Configured	
EBX	31	22	10	W	CPUID_CONFIG		As Configured	
ECX	31	0	32	Number of Sets	CPUID_CONFIG		As Configured	
EDX	0	0	1	WBINVD	CPUID_CONFIG		As Configured	
EDX	1	1	1	Cache Inclusiveness	CPUID_CONFIG		As Configured	
EDX	2	2	1	Reserved_31_2	CPUID_CONFIG		As Configured	
EDX	31	3	29	0	CPUID_CONFIG		As Configured	
Leaf 0x4 / Sub-Leaf 0x1								
EAX	4	0	5	Type	CPUID_CONFIG		As Configured	
EAX	7	5	3	Level	CPUID_CONFIG		As Configured	
EAX	8	8	1	Self Initializing	CPUID_CONFIG		As Configured	
EAX	9	9	1	Fully Associative	CPUID_CONFIG		As Configured	
EAX	13	10	4	Reserved	CPUID_CONFIG		As Configured	
EAX	25	14	12	Addressable IDs Sharing this Cache	CPUID_CONFIG		As Configured	
EAX	31	26	6	Addressable IDs for Cores in Package	CPUID_CONFIG		As Configured	
EBX	11	0	12	L	N/A		Native	
EBX	21	12	10	P	CPUID_CONFIG		As Configured	
EBX	31	22	10	W	CPUID_CONFIG		As Configured	
ECX	31	0	32	Number of Sets	CPUID_CONFIG		As Configured	
EDX	0	0	1	WBINVD	CPUID_CONFIG		As Configured	
EDX	1	1	1	Cache Inclusiveness	CPUID_CONFIG		As Configured	
EDX	2	2	1	Reserved_31_2	CPUID_CONFIG		As Configured	
EDX	31	3	29	0	CPUID_CONFIG		As Configured	
Leaf 0x4 / Sub-Leaf 0x2								
EAX	4	0	5	Type	CPUID_CONFIG		As Configured	
EAX	7	5	3	Level	CPUID_CONFIG		As Configured	
EAX	8	8	1	Self Initializing	CPUID_CONFIG		As Configured	
EAX	9	9	1	Fully Associative	CPUID_CONFIG		As Configured	
EAX	13	10	4	Reserved_13_10	CPUID_CONFIG		As Configured	
EAX	25	14	12	Addressable IDs Sharing this Cache	CPUID_CONFIG		As Configured	
EAX	31	26	6	Addressable IDs for Cores in Package	CPUID_CONFIG		As Configured	
EBX	11	0	12	L	N/A		Native	
EBX	21	12	10	P	CPUID_CONFIG		As Configured	
EBX	31	22	10	W	CPUID_CONFIG		As Configured	
ECX	31	0	32	Number of Sets	CPUID_CONFIG		As Configured	
EDX	0	0	1	WBINVD	CPUID_CONFIG		As Configured	
EDX	1	1	1	Cache Inclusiveness	CPUID_CONFIG		As Configured	
EDX	2	2	1	Reserved_31_2	CPUID_CONFIG		As Configured	
EDX	31	3	29	0	CPUID_CONFIG		As Configured	
Leaf 0x4 / Sub-Leaf 0x3								
EAX	4	0	5	Type	CPUID_CONFIG		As Configured	
EAX	7	5	3	Level	CPUID_CONFIG		As Configured	
EAX	8	8	1	Self Initializing	CPUID_CONFIG		As Configured	
EAX	9	9	1	Fully Associative	CPUID_CONFIG		As Configured	
EAX	13	10	4	Reserved_13_10	CPUID_CONFIG		As Configured	
EAX	25	14	12	Addressable IDs Sharing this Cache	CPUID_CONFIG		As Configured	
EAX	31	26	6	Addressable IDs for Cores in Package	CPUID_CONFIG		As Configured	
EBX	11	0	12	L	N/A		Native	
EBX	21	12	10	P	CPUID_CONFIG		As Configured	
EBX	31	22	10	W	CPUID_CONFIG		As Configured	
ECX	31	0	32	Number of Sets	CPUID_CONFIG		As Configured	
EDX	0	0	1	WBINVD	CPUID_CONFIG		As Configured	
EDX	1	1	1	Cache Inclusiveness	CPUID_CONFIG		As Configured	
EDX	2	2	1	Complex cache indexing	CPUID_CONFIG		As Configured	
EDX	31	3	29	Reserved_31_3	CPUID_CONFIG		As Configured	
Leaf 0x4 / Sub-Leaf 0x4								
EAX	4	0	5	Type	N/A		Fixed	0x0
EAX	7	5	3	Level	N/A		Fixed	0x0
EAX	8	8	1	Self Initializing	N/A		Fixed	0x0
EAX	9	9	1	Fully Associative	N/A		Fixed	0x0
EAX	13	10	4	Reserved	N/A		Fixed	0x0
EAX	25	14	12	Addressable IDs Sharing this Cache	N/A		Fixed	0x0
EAX	31	26	6	Addressable IDs for Cores in Package	N/A		Fixed	0x0
EBX	11	0	12	L	N/A		Fixed	0x0
EBX	21	12	10	P	N/A		Fixed	0x0
EBX	31	22	10	W	N/A		Fixed	0x0
ECX	31	0	32	Number of Sets	N/A		Fixed	0x0
EDX	0	0	1	WBINVD	N/A		Fixed	0x0

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
EDX	1	1	1	Cache Inclusiveness	N/A		Fixed	0x0
EDX	2	2	1	Complex Cache Indexing	N/A		Fixed	0x0
EDX	31	3	29	Reserved	N/A		Fixed	0x0
Leaf 0x7 / Sub-Leaf 0x0								
EAX	31	0	32	Max Sub-Leaves	N/A		Fixed	0x1
EBX	0	0	1	FSGSBASE	N/A		Fixed	0x1
EBX	1	1	1	IA32_TSC_ADJUST	N/A		Fixed	0x0
EBX	2	2	1	SGX	N/A		Fixed	0x0
EBX	3	3	1	BMI1	CPUID_CONFIG		As Configured (if Native)	
EBX	4	4	1	HLE	N/A		Native	
EBX	5	5	1	AVX2	XFAM	XFAM[2]	As Configured (if Native)	
EBX	6	6	1	FDP_EXCPTN_ONLY	N/A		Native	
EBX	7	7	1	SMEP	N/A		Native	
EBX	8	8	1	BMI2	CPUID_CONFIG		As Configured (if Native)	
EBX	9	9	1	Enhanced REP MOVSB/STOSB	N/A		Native	
EBX	10	10	1	INVPCID	N/A		Native	
EBX	11	11	1	RTM	N/A		Fixed	0x1
EBX	12	12	1	PQM	CPUID_CONFIG		As Configured (if Native)	
EBX	13	13	1	FCS/FDS Deprecation	N/A		Native	
EBX	14	14	1	MPX	N/A		Fixed	0x0
EBX	15	15	1	Cache QoS Enforcement	CPUID_CONFIG		As Configured (if Native)	
EBX	16	16	1	AVX512F	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	17	17	1	AVX512DQ	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	18	18	1	RDSEED	N/A		Fixed	0x1
EBX	19	19	1	ADCX/ADOX	CPUID_CONFIG		As Configured (if Native)	
EBX	20	20	1	SMAP/CLAC/STAC	N/A		Fixed	0x1
EBX	21	21	1	AVX512_IFMA	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	22	22	1	PCOMMIT	N/A		Native	
EBX	23	23	1	CLFLUSHOPT	N/A		Fixed	0x1
EBX	24	24	1	CLWB	N/A		Fixed	0x1
EBX	25	25	1	RTIT	XFAM	XFAM[8]	As Configured (if Native)	
EBX	26	26	1	AVX512PF	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	27	27	1	AVX512ER	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	28	28	1	AVX512CD	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	29	29	1	SHA	N/A		Fixed	0x1
EBX	30	30	1	AVX512BW	XFAM	XFAM[7:5]	As Configured (if Native)	
EBX	31	31	1	AVX512VL	XFAM	XFAM[7:5]	As Configured (if Native)	
ECX	0	0	1	PREFETCHWT1	N/A		Native	
ECX	1	1	1	AVX512VBMI	XFAM	XFAM[7:5]	As Configured (if Native)	
ECX	2	2	1	UMIP	N/A		Native	
ECX	3	3	1	PKU	XFAM	XFAM[9]	As Configured (if Native)	
ECX	4	4	1	OSPKE	N/A		Calculated	CR4.PKE
ECX	5	5	1	MONITORX/MWAITX	CPUID_CONFIG		As Configured (if Native)	
ECX	6	6	1	AVX512_VBMI2	XFAM	XFAM[7:5]	As Configured (if Native)	
ECX	7	7	1	CET Shadow Stack	XFAM	XFAM[12:11]	As Configured (if Native)	
ECX	8	8	1	GFNI	N/A		Native	
ECX	9	9	1	VAES	XFAM	XFAM[2]	As Configured (if Native)	
ECX	10	10	1	VPCLMULQDQ	XFAM	XFAM[2]	As Configured (if Native)	
ECX	11	11	1	AVX512_VNNI	XFAM	XFAM[7:5]	As Configured (if Native)	
ECX	12	12	1	AVX512_BITALG	XFAM	XFAM[7:5]	As Configured (if Native)	
ECX	13	13	1	TME	CPUID_CONFIG		As Configured (if Native)	
ECX	14	14	1	AVX512_VPOPCNTDQ	XFAM	XFAM[7:5]	As Configured (if Native)	
ECX	15	15	1	FZM	N/A		Fixed	0x0
ECX	16	16	1	57 bit Address Support	N/A		Native	
ECX	21	17	5	MAWAU for MPX	N/A		Fixed	0x0
ECX	22	22	1	RDPID	N/A		Native	
ECX	23	23	1	KL_ENABLED	ATTRIBUTES	KL	As Configured (if Native)	
ECX	24	24	1	BUSLOCK	N/A		Fixed	0x1
ECX	25	25	1	CLDEMOTE	N/A		Native	
ECX	26	26	1	Reserved_26	N/A		Native	
ECX	27	27	1	MOVDIRI	N/A		Native	
ECX	28	28	1	MOVDIR64B	N/A		Fixed	0x1
ECX	29	29	1	ENQCMD	N/A		Fixed	0x0
ECX	30	30	1	SGX_LC	N/A		Fixed	0x0
ECX	31	31	1	PKS	ATTRIBUTES	PKS	As Configured (if Native)	
EDX	0	0	1	Reserved_0	N/A		Fixed	0x0
EDX	1	1	1	Reserved_1	N/A		Fixed	0x0
EDX	2	2	1	AVX512_4VNNIW	XFAM	XFAM[7:5]	As Configured (if Native)	

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
EDX	3	3	1	AVX512_4FMAPS	XFAM	XFAM[7:5]	As Configured (if Native)	
EDX	4	4	1	Fast Short REP MOV	N/A		Native	
EDX	5	5	1	ULI	XFAM	XFAM[14]	As Configured (if Native)	
EDX	6	6	1	Reserved_6	N/A		Fixed	0x0
EDX	7	7	1	Reserved_7	N/A		Fixed	0x0
EDX	8	8	1	AVX512_VP2INTERSECT	XFAM	XFAM[7:5]	As Configured (if Native)	
EDX	9	9	1	Reserved_9	N/A		Fixed	0x0
EDX	10	10	1	MD_CLEAR supported	N/A		Native	
EDX	11	11	1	Reserved_11	N/A		Fixed	0x0
EDX	12	12	1	Reserved_12	N/A		Fixed	0x0
EDX	13	13	1	Reserved_13	N/A		Fixed	0x0
EDX	14	14	1	SERIALIZE Inst	N/A		Native	
EDX	15	15	1	Hetero Part	N/A		Native	
EDX	16	16	1	TSXLDTRK	N/A		Native	
EDX	17	17	1	Reserved_17	N/A		Fixed	0x0
EDX	18	18	1	PCONFIG	CPUID_CONFIG		As Configured (if Native)	
EDX	19	19	1	Architectural LBR support	XFAM	XFAM[15]	As Configured (if Native)	
EDX	20	20	1	CET	XFAM	XFAM[12:11]	As Configured (if Native)	
EDX	21	21	1	Reserved_21	N/A		Fixed	0x0
EDX	22	22	1	TMUL_AMX-BF16	XFAM	XFAM[18:17]	As Configured (if Native)	
EDX	23	23	1	FP16	XFAM	XFAM[7:5]	As Configured (if Native)	
EDX	24	24	1	TMUL_AMX-TILE	XFAM	XFAM[18:17]	As Configured (if Native)	
EDX	25	25	1	TMUL_AMX-INT8	XFAM	XFAM[18:17]	As Configured (if Native)	
EDX	26	26	1	IBRS (indirect branch restricted speculation)	N/A		Fixed	0x1
EDX	27	27	1	STIBP (single thread indirect branch predictors)	N/A		Native	
EDX	28	28	1	L1D_FLUSH. IA32_FLUSH_CMD support.	N/A		Native	
EDX	29	29	1	IA32_ARCH_CAPABILITIES Support	N/A		Fixed	0x1
EDX	30	30	1	IA32_CORE_CAPABILITIES Present	N/A		Fixed	0x1
EDX	31	31	1	SSBD (Speculative Store Bypass Disable)	N/A		Fixed	0x1
Leaf 0x7 / Sub-Leaf 0x1								
EAX	2	0	3	Reserved_3_0	N/A		Fixed	0x0
EAX	3	3	1	Reserved_4	N/A		Fixed	0x0
EAX	4	4	1	VEX VNNI	XFAM	XFAM[7:5]	As Configured (if Native)	
EAX	5	5	1	AVX512_BF16	XFAM	XFAM[7:5]	As Configured (if Native)	
EAX	6	6	1	Reserved_6	N/A		Fixed	0x0
EAX	7	7	1	Reserved_7	N/A		Fixed	0x0
EAX	8	8	1	Reserved_8	N/A		Fixed	0x0
EAX	9	9	1	Reserved_9	N/A		Fixed	0x0
EAX	10	10	1	Fast Zero-Length MOVSB	N/A		Native	
EAX	11	11	1	Fast Short STOSB	N/A		Native	
EAX	12	12	1	Fast short CMPSB/SCASB	N/A		Native	
EAX	21	13	9	Reserved_21_13	N/A		Fixed	0x0
EAX	22	22	1	HRESET	N/A		Fixed	0x0
EAX	23	23	1	Reserved_23	N/A		Fixed	0x0
EAX	24	24	1	Reserved_24	N/A		Fixed	0x0
EAX	31	25	7	Reserved_31_25	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0x8								
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0xa								
EAX	7	0	8	Version	ATTRIBUTES	PERFMON	As Configured (if Native)	
EAX	15	8	8	Number of GP Counters	ATTRIBUTES	PERFMON	As Configured (if Native)	
EAX	23	16	8	Width of GP Counters	ATTRIBUTES	PERFMON	As Configured (if Native)	
EAX	31	24	8	Length of EBX Vector	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	0	0	1	Core Cycles Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	1	1	1	Instructions Retired Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	2	2	1	Reference Cycles Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	3	3	1	Last-Level Cache References Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	4	4	1	Last-Level Cache Misses Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	5	5	1	Branch Instruction Retired Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	6	6	1	Branch Mispredict Retired Not Available	ATTRIBUTES	PERFMON	As Configured (if Native)	
EBX	31	7	25	Reserved	ATTRIBUTES	PERFMON	As Configured (if Native)	

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
ECX	3	0	4	Fixed Counter Support	ATTRIBUTES	PERFMON	As Configured (if Native)	
ECX	31	4	28	Reserved	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	4	0	5	Number of Fixed-Function Counters	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	12	5	8	Width of Fixed-Function Counters	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	13	13	1	Reserved_13	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	14	14	1	Reserved_14	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	15	15	1	AnyThread Deprecation	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	16	16	1	TopDown Support	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	23	17	7	Reserved_23_17	ATTRIBUTES	PERFMON	As Configured (if Native)	
EDX	31	24	8	Bit Vector Length	ATTRIBUTES	PERFMON	As Configured (if Native)	
Leaf 0xd / Sub-Leaf 0x0								
EAX	0	0	1	X87	N/A		Fixed	0x1
EAX	1	1	1	SSE	N/A		Fixed	0x1
EAX	2	2	1	AVX256	XFAM	XFAM[2]	As Configured (if Native)	
EAX	3	3	1	PL_BNDREGS	N/A		Fixed	0x0
EAX	4	4	1	PL_BNDCFS	N/A		Fixed	0x0
EAX	5	5	1	KMASK	XFAM	XFAM[7:5]	As Configured (if Native)	
EAX	6	6	1	AVX3 ZMM 15:0	XFAM	XFAM[7:5]	As Configured (if Native)	
EAX	7	7	1	AVX3 ZMM 31:18	XFAM	XFAM[7:5]	As Configured (if Native)	
EAX	8	8	1	Reserved_8	N/A		Fixed	0x0
EAX	9	9	1	PKRU	XFAM	XFAM[9]	As Configured (if Native)	
EAX	16	10	7	Reserved_16_10	N/A		Fixed	0x0
EAX	17	17	1	AMX - XTILECFG	XFAM	XFAM[18:17]	As Configured (if Native)	
EAX	18	18	1	AMX - XTILEDATA	XFAM	XFAM[18:17]	As Configured (if Native)	
EAX	31	19	13	Reserved_31_19	N/A		Fixed	0x0
EBX	31	0	32	Max Bytes for Enabled Features	N/A		Calculated	Native
ECX	31	0	32	Max Bytes for Supported Features	XFAM		As Configured	
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0xd / Sub-Leaf 0x1								
EAX	0	0	1	Supports XSAVEOPT	N/A		Fixed	0x1
EAX	1	1	1	Supports XSAVEC and compacted XRSTOR	N/A		Fixed	0x1
EAX	2	2	1	Supports XGETBV with ECX = 1	N/A		Native	
EAX	3	3	1	Supports XSAVES/XRSTORS and IA32_XSS	N/A		Fixed	0x1
EAX	4	4	1	XFD support	XFAM		As Configured	
EAX	31	5	27	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Max Bytes for Enabled Features	N/A		Calculated	Native
ECX	7	0	8	reserved_7_0	N/A		Fixed	0x0
ECX	8	8	1	XSS_RTIT	XFAM	XFAM[8]	As Configured (if Native)	
ECX	9	9	1	reserved_9	N/A		Fixed	0x0
ECX	10	10	1	PASID	N/A		Fixed	0x0
ECX	11	11	1	U_CET	XFAM	XFAM[12:11]	As Configured (if Native)	
ECX	12	12	1	S_CET	XFAM	XFAM[12:11]	As Configured (if Native)	
ECX	13	13	1	HDC	N/A		Fixed	0x0
ECX	14	14	1	ULI/UNIT	XFAM	XFAM[14]	As Configured (if Native)	
ECX	15	15	1	XSS_ARCH_LBRS	XFAM	XFAM[15]	As Configured (if Native)	
ECX	16	16	1	HWP Request	N/A		Fixed	0x0
ECX	31	17	15	Reserved_31_17	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0xd / Sub-Leaves 0x2-0x12								
EAX	31	0	32	Size	XFAM	XFAM[n]	As Configured (if Native)	
EBX	31	0	32	Offset	XFAM	XFAM[n]	As Configured (if Native)	
ECX	0	0	1	IA32_XSS	XFAM	XFAM[n]	As Configured (if Native)	
ECX	1	1	1	0	XFAM	XFAM[n]	As Configured (if Native)	
ECX	31	2	30	0	XFAM	XFAM[n]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[n]	As Configured (if Native)	
Leaf 0xe								
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0x11								
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
Leaf 0x12								
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
				Leaf 0x13				
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
				Leaf 0x14 / Sub-Leaf 0x0				
EAX	31	0	32	Max Valid Subleaf	XFAM	XFAM[8]	As Configured (if Native)	
EBX	0	0	1	CR3 Filtering	XFAM	XFAM[8]	As Configured (if Native)	
EBX	1	1	1	Cycle Accurate Mode	XFAM	XFAM[8]	As Configured (if Native)	
EBX	2	2	1	IP Filtering	XFAM	XFAM[8]	As Configured (if Native)	
EBX	3	3	1	MSRs Preserved Across Warm Reset	XFAM	XFAM[8]	As Configured (if Native)	
EBX	4	4	1	PTWRITE Support	XFAM	XFAM[8]	As Configured (if Native)	
EBX	5	5	1	Power Event Trace Support	XFAM	XFAM[8]	As Configured (if Native)	
EBX	6	6	1	PSB/PMI Injection Support	XFAM	XFAM[8]	As Configured (if Native)	
EBX	7	7	1	PT Event Trace Support	XFAM	XFAM[8]	As Configured (if Native)	
EBX	31	8	24	Reserved	XFAM	XFAM[8]	As Configured (if Native)	
ECX	0	0	1	ToPA Output Supported	XFAM	XFAM[8]	As Configured (if Native)	
ECX	1	1	1	ToPA Tables Support Multiple Regions	XFAM	XFAM[8]	As Configured (if Native)	
ECX	2	2	1	Single-Range Output Supported	XFAM	XFAM[8]	As Configured (if Native)	
ECX	30	3	28	Reserved	XFAM	XFAM[8]	As Configured (if Native)	
ECX	31	31	1	IP Payload Contains LIP	XFAM	XFAM[8]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[8]	As Configured (if Native)	
				Leaf 0x14 / Sub-Leaf 0x1				
EAX	1	0	2	MTC Period Options	XFAM	XFAM[8]	As Configured (if Native)	
EAX	15	2	14	Reserved	XFAM	XFAM[8]	As Configured (if Native)	
EAX	31	16	16	Number of Address Ranges Supported	XFAM	XFAM[8]	As Configured (if Native)	
EBX	15	0	16	Cycle Thresholds	XFAM	XFAM[8]	As Configured (if Native)	
EBX	31	16	16	PSB Frequencies	XFAM	XFAM[8]	As Configured (if Native)	
ECX	31	0	32	Reserved	XFAM	XFAM[8]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[8]	As Configured (if Native)	
				Leaf 0x15				
EAX	31	0	32	Denominator	N/A		Fixed	0x1
EBX	31	0	32	Numerator	Other	TSC_FREQUENCY	As Configured	
ECX	31	0	32	Nominal ART Frequency	N/A		Fixed	0x017D7840
EDX	31	0	32	Reserved	N/A		Fixed	0x0
				Leaf 0x19				
EAX	0	0	1	CPL0 Restriction	ATTRIBUTES	KL	As Configured (if Native)	
EAX	1	1	1	Decrypt only Restriction	ATTRIBUTES	KL	As Configured (if Native)	
EAX	2	2	1	Encrypt only Restriction	ATTRIBUTES	KL	As Configured (if Native)	
EAX	3	3	1	Process Restriction	ATTRIBUTES	KL	As Configured (if Native)	
EAX	31	4	28	Reserved_31_4	ATTRIBUTES	KL	As Configured (if Native)	
EBX	0	0	1	AES KL Enabled	ATTRIBUTES	KL	As Configured (if Native)	
EBX	1	1	1	Reserved	ATTRIBUTES	KL	As Configured (if Native)	
EBX	2	2	1	AES wide KL Support	ATTRIBUTES	KL	As Configured (if Native)	
EBX	3	3	1	Reserved	ATTRIBUTES	KL	As Configured (if Native)	
EBX	4	4	1	IW Key Backup Support	ATTRIBUTES	KL	As Configured (if Native)	
EBX	31	5	27	Reserved	ATTRIBUTES	KL	As Configured (if Native)	
ECX	0	0	1	LOADIWKEY No Backup parameter Support	ATTRIBUTES	KL	As Configured (if Native)	
ECX	1	1	1	Random IWKey Support	N/A		Fixed	0x0
ECX	31	2	30	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
				Leaf 0x1c				
EAX	7	0	8	Supported LBR depth values	XFAM	XFAM[15]	As Configured (if Native)	
EAX	29	8	22	Reserved_29_8	XFAM	XFAM[15]	As Configured (if Native)	
EAX	30	30	1	Deep C-state May Reset	XFAM	XFAM[15]	As Configured (if Native)	
EAX	31	31	1	IP values contain LIP	XFAM	XFAM[15]	As Configured (if Native)	
EBX	0	0	1	CPL Filtering Supported	XFAM	XFAM[15]	As Configured (if Native)	
EBX	1	1	1	Branch Filtering Supported	XFAM	XFAM[15]	As Configured (if Native)	
EBX	2	2	1	Call-stack Mode Supported	XFAM	XFAM[15]	As Configured (if Native)	
EBX	31	3	29	Reserved_31_3	XFAM	XFAM[15]	As Configured (if Native)	
ECX	0	0	1	Mispredict Bit Supported	XFAM	XFAM[15]	As Configured (if Native)	
ECX	1	1	1	Timed LBRs Supported	XFAM	XFAM[15]	As Configured (if Native)	
ECX	2	2	1	Branch Type Field Supported	XFAM	XFAM[15]	As Configured (if Native)	
ECX	31	3	29	Reserved_31_3	XFAM	XFAM[15]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[15]	As Configured (if Native)	
				Leaf 0x1d / Sub-Leaf 0x0				

CPUID Field					Configuration by TDH.MNG.INIT		Virtualization	
Reg.	MSB	LSB	Field Size	Field Name	TD_PARAMS Section	Configuration Details	Virtualization Type	Virtualization Details
EAX	0	0	1	TILE support	XFAM	XFAM[18:17]	As Configured (if Native)	
EAX	31	1	31	Reserved_31_1	XFAM	XFAM[18:17]	As Configured (if Native)	
EBX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
ECX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
				Leaf 0x1d / Sub-Leaf 0x1				
EAX	15	0	16	total_tile_bytes	XFAM	XFAM[18:17]	As Configured (if Native)	
EAX	31	16	16	bytes_per_tile	XFAM	XFAM[18:17]	As Configured (if Native)	
EBX	15	0	16	bytes_per_row	XFAM	XFAM[18:17]	As Configured (if Native)	
EBX	31	16	16	max_names	XFAM	XFAM[18:17]	As Configured (if Native)	
ECX	15	0	16	max_rows	XFAM	XFAM[18:17]	As Configured (if Native)	
ECX	31	16	16	Reserved_31_16	XFAM	XFAM[18:17]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
				Leaf 0x1e				
EAX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
EBX	7	0	8	impl.tmul_maxk (rows or cols)	XFAM	XFAM[18:17]	As Configured (if Native)	
EBX	23	8	16	impl.tmul_maxn (column bytes)	XFAM	XFAM[18:17]	As Configured (if Native)	
EBX	31	24	8	Reserved_31_24	XFAM	XFAM[18:17]	As Configured (if Native)	
ECX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
EDX	31	0	32	Reserved	XFAM	XFAM[18:17]	As Configured (if Native)	
				Leaf 0x20				
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
				Leaf 0x21 / Sub-Leaf 0x0				
EAX	31	0	32	Maximum sub-leaf	N/A		Fixed	0x00000000
EBX	31	0	32	“Inte”	N/A		Fixed	0x65746E49
ECX	31	0	32	“ “	N/A		Fixed	0x20202020
EDX	31	0	32	“ITDX”	N/A		Fixed	0x5844546C
				Leaf 0x80000000				
EAX	31	0	32	MaxIndex	N/A		Native	
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0
				Leaf 0x80000001				
EAX	31	0	32	Reserved	N/A		Fixed	0x0
EBX	31	0	32	Reserved	N/A		Fixed	0x0
ECX	0	0	1	LAHF/SAHF in 64-bit Mode	N/A		Native	
ECX	4	1	4	Reserved_4_1	N/A		Fixed	0x0
ECX	5	5	1	LZCNT	N/A		Native	
ECX	7	6	2	Reserved_7_6	N/A		Fixed	0x0
ECX	8	8	1	PREFETCHW	N/A		Native	
ECX	31	9	23	Reserved_31_9	N/A		Fixed	0x0
EDX	10	0	11	Reserved_10_0	N/A		Fixed	0x0
EDX	11	11	1	SYSCALL/SYSRET in 64-bit Mode	N/A		Native	
EDX	19	12	8	Reserved_19_12	N/A		Fixed	0x0
EDX	20	20	1	Execute Disable Bit	N/A		Fixed	0x1
EDX	25	21	5	Reserved_25_21	N/A		Fixed	0x0
EDX	26	26	1	1GB Pages	N/A		Fixed	0x1
EDX	27	27	1	RDTSCP and IA32_TSC_AUX	N/A		Fixed	0x1
EDX	28	28	1	Reserved_28	N/A		Fixed	0x0
EDX	29	29	1	Intel 64	N/A		Fixed	0x1
EDX	31	30	2	Reserved_31_30	N/A		Fixed	0x0
				Leaf 0x80000008				
EAX	7	0	8	Number of Physical Address Bits	N/A		Fixed	0x34
EAX	15	8	8	Number of Linear Address Bits	N/A		Native	
EAX	31	16	16	Reserved	N/A		Fixed	0x0
EBX	8	0	9	Reserved_8_0	N/A		Fixed	0x0
EBX	9	9	1	WBNOINVD support	N/A		Fixed	0x1
EBX	31	10	22	Reserved_31_10	N/A		Fixed	0x0
ECX	31	0	32	Reserved	N/A		Fixed	0x0
EDX	31	0	32	Reserved	N/A		Fixed	0x0

19.ABI Reference: Constants

This chapter describes the constants designed to be used in the Intel TDX module.

19.1. Interface Function Completion Status Codes

Note: This section provides a high-level overview of function completion status, as defined. Implementation details may differ.

This section defines the function completion status codes. The structure of the status codes is described in 17.3.2. Three tables are provided below: class table, code table and operand ID table.

19.1.1. Function Completion Status Code Classes (Bits 47:40)

Table 19.1: Function Completion Status Code Classes (Bits 47:40) Definition

Class ID	Class Name	Description
0	General	General function completion status
1	Invalid Operand	An invalid operand value has been provided, e.g., HKID is out of range, HPA overlaps SEAMRR, GPA is not private, etc.
2	Resource Busy	Resource is busy, there is a concurrency conflict.
3	Page Metadata	Page metadata (in PAMT) are incorrect, e.g., page type is wrong.
4	Dependent Resources	The state of dependent resources is incorrect, e.g., there are TD pages while trying to reclaim a TDR page.
5	Intel TDX Module State	The Intel TDX module state is incorrect.
6	TD State	The state of the TD is incorrect, e.g., it has not been initialized yet.
7	TD VCPU State	The state of the TD VCPU is incorrect, e.g., it is corrupted.
8	Key Management	The status code is related to key management, e.g., keys are not configured.
9	Platform	The status code is related to platform configuration or state.
10	Physical Memory	The status code is related to physical memory.
11	Guest TD Memory	The status code is related to guest TD memory.
255	Reserved	Reserved for use by host VMM or guest TD software This value is never used by the TDX module.

19.1.2. Function Completion Status Codes

Table 19.2: Function Completion Status Codes Definition

Error and Recoverability Flags, Class and Name (Bits 63:32)			Details L2 (Bits 31:0)	Description
Value (Hex)	Status	Name		
0x00000000	Success	TDX_SUCCESS	TDH.VP.ENTER: Exit Reason Other : 0	Function completed successfully.
0x40000001	Non-Recover.	TDX_NON_RECOVERABLE_VCPU	TDH.VP.ENTER: Exit Reason	TD exit due to a non-recoverable VCPU state (e.g., triple fault) – VCPU is disabled
0x40000002	Non-Recover.	TDX_NON_RECOVERABLE_TD	TDH.VP.ENTER: Exit Reason	TD exit due to a non-recoverable TD state – TD is disabled
0x80000003	Recover. Error	TDX_INTERRUPTED_RESUMABLE	0	Function operation has been interrupted by an external event, and it may be resumed from the point it was interrupted by calling it again.
0x80000004	Recover. Error	TDX_INTERRUPTED_RESTARTABLE	0	Function operation has been interrupted by an external event, and it may be restarted (from its beginning) by calling it again.
0x40000005	Non-Recover.	TDX_NON_RECOVERABLE_TD_FATAL	TDH.VP.ENTER: Exit Reason	TD exit due to a fatal TD state (e.g., machine check caused by a memory integrity check error) – TD is disabled and its private memory can't be accessed.

Error and Recoverability Flags, Class and Name (Bits 63:32)			Details L2 (Bits 31:0)	Description
Value (Hex)	Status	Name		
0xC0000006	Error	TDX_INVALID_RESUMPTION	0	Resumed function in invalid, e.g., its operands are different than the last interrupted function.
0xC0000007	Error	TDX_NON_RECOVERABLE_TD_NO_APIC	TDH.VP.ENTER: Exit Reason	TD is running with local APIC disabled
0xC0000100	Error	TDX_OPERAND_INVALID	Operand ID	Operand is invalid.
0xC0000101	Error	TDX_OPERAND_ADDR_RANGE_ERROR	Operand ID	Operand address is out of range (e.g., not in a TDMR).
0x80000200	Recover. Error	TDX_OPERAND_BUSY	Operand ID	The operand is busy (e.g., it is locked in Exclusive mode).
0x80000201	Recover. Error	TDX_PREVIOUS_TLB_EPOCH_BUSY	0	TDH.MEM.TRACK failed because one or more of the TD's VCPUs are running, and their VCPU epoch is the previous TD epoch.
0x80000202	Recover. Error	TDX_SYS_BUSY	0	The operation was invoked when another TDX module operation was in progress.
0xC0000300	Error	TDX_PAGE_METADATA_INCORRECT	Operand ID	Physical page metadata (in PAMT) are incorrect for the requested operation.
0x00000301	Success	TDX_PAGE_ALREADY_FREE	Operand ID	Physical page is already marked as PT_FREE.
0xC0000302	Error	TDX_PAGE_NOT_OWNED_BY_TD	Operand ID	Physical page PAMT entry's OWNER field does not point to the TD's TDR page
0xC0000303	Error	TDX_PAGE_NOT_FREE	Operand ID	Physical page is not free
0xC0000400	Error	TDX_TD_ASSOCIATED_PAGES_EXIST	0	Physical pages associated with the TD exist in memory.
0xC0000500	Error	TDX_SYSINIT_NOT_PENDING	0	Attempting TDH.SYS.INIT when not expected.
0xC0000501	Error	TDX_SYSINIT_NOT_DONE	0	Attempting non-TDH.SYS.INIT SEAMCALL leaf before TDH.SYS.INIT was done.
0xC0000502	Error	TDX_SYSINITLP_NOT_DONE	0	Attempting non-TDH.SYS.LP.INIT SEAMCALL leaf before TDH.SYS.LP.INIT was done on this LP.
0xC0000503	Error	TDX_SYSINITLP_DONE	0	Attempting TDH.SYS.LP.INIT when already done on this LP.
0xC0000505	Error	TDX_SYS_NOT_READY	0	Attempting to execute a non-initialization SEAMCALL function before initialization sequence completed.
0xC0000506	Error	TDX_SYS_SHUTDOWN	0	Attempting to execute SEAMCALL when the Intel TDX module is being shut down.
0xC0000507	Error	TDX_SYS_KEY_CONFIG_NOT_PENDING	0	Attempting TDH.SYS.KEY.CONFIG when it is not pending.
0xC0000600	Error	TDX_TD_NOT_INITIALIZED	0	TD has not been initialized (by TDH.MNG.INIT).
0xC0000601	Error	TDX_TD_INITIALIZED	0	TD has been initialized (by TDH.MNG.INIT).
0xC0000602	Error	TDX_TD_NOT_FINALIZED	0	TD measurement has not been finalized (by TDH.MR.FINALIZE).
0xC0000603	Error	TDX_TD_FINALIZED	0	TD measurement has been finalized (by TDH.MR.FINALIZE).
0xC0000604	Error	TDX_TD_FATAL	0	TD is in a FATAL error state.
0xC0000605	Error	TDX_TD_NON_DEBUG	0	TD's ATTRIBUTES.DEBUG bit is 0.
0xC0000607	Error	TDX_LIFECYCLE_STATE_INCORRECT	0	The TD's LIFECYCLE_STATE is incorrect for the required operation.
0xC0000610	Error	TDX_TDCX_NUM_INCORRECT	0	The number of TDCX pages is incorrect.
0xC0000700	Error	TDX_VCPU_STATE_INCORRECT	0	The VCPU state is incorrect for the requested operation.
0x80000701	Recover. Error	TDX_VCPU_ASSOCIATED	0	The VCPU is already associated with another LP.
0x80000702	Recover. Error	TDX_VCPU_NOT_ASSOCIATED	0	The VCPU is not associated with the current LP.
0xC0000703	Error	TDX_TDVPX_NUM_INCORRECT	0	The number of TDVPX pages is incorrect.
0xC0000704	Error	TDX_NO_VALID_VE_INFO	0	There is no valid #VE information.
0xC0000705	Error	TDX_MAX_VCPUS_EXCEEDED	0	TD's maximum number of VCPUs has been exceeded.
0xC0000706	Error	TDX_TSC_ROLLBACK	0	Time Stamp Counter value is lower than on last TD exit.
0xC0000720	Error	TDX_FIELD_NOT_WRITABLE	0	Field code and write mask are for a read-only field.
0xC0000721	Error	TDX_FIELD_NOT_READABLE	0	Field code is for an unreadable field.
0xC0000730	Error	TDX_TD_VMCS_FIELD_NOT_INITIALIZED	Bits 31:0: VMCS field code	The TD VMCS field has not been initialized.
0x80000800	Recover. Error	TDX_KEY_GENERATION_FAILED	0	Failed to generate a random key. This is typically caused by an entropy error of the CPU's random number generator, and may be impacted by RDSEED, RDRAND or PCONFIG executing on other LPs. The operation should be retried.
0x80000810	Recover. Error	TDX_TD_KEYS_NOT_CONFIGURED	0	TD keys have not been configured on the hardware.
0xC0000811	Error	TDX_KEY_STATE_INCORRECT	0	TDCS.KEY_STATE is incorrect for the required operation.
0x00000815	Success	TDX_KEY_CONFIGURED	0	The key is already configured on the current package.
0x80000817	Recover. Error	TDX_WBCACHE_NOT_COMPLETE	0	Attempting to execute TDH.MNG.KEY.FREEID when TDH.PHYMEM.CACHE.WB has not completed its operation.
0xC0000820	Error	TDX_HKID_NOT_FREE	0	A provided HKID cannot be assigned because it is not free.
0x00000821	Success	TDX_NO_HKID_READY_TO_WBCACHE	0	No private HKID is in the HKID_FLUSHED state, ready for TDH.PHYMEM.CACHE.WB.

Error and Recoverability Flags, Class and Name (Bits 63:32)			Details L2 (Bits 31:0)	Description
Value (Hex)	Status	Name		
0xC0000823	Error	TDX_WBCACHE_RESUME_ERROR	0	Resume of a previously-interrupted function has been aborted due to wrong HKID.
0x80000824	Recover. Error	TDX_FLUSHVP_NOT_DONE	0	TDH.VP.FLUSH was not done on all required VCPUs; some VCPUs are still associated with LPs.
0xC0000825	Error	TDX_NUM_ACTIVATED_HKIDS_NOT_SUPPORTED	Bits 31:0: Maximum supported HKIDs	The number of activated key IDs on the platform is not supported.
0xC0000900	Error	TDX_INCORRECT_CPUID_VALUE	0	A CPUID value is incorrect.
0xC0000901	Error	TDX_BOOT_NT4_SET	0	MSR IA32_MISC_ENABLES bit 22 (Boot NT4) is set.
0xC0000902	Error	TDX_INCONSISTENT_CPUID_FIELD	0	A field returned by CPUID is inconsistent between LPs.
0xC0000903	Error	TDX_CPUID_MAX_SUBLEAVES_UNRECOGNIZED	CPUID leaf	The maximum number of sub-leaves for this CPUID leaf is not recognized.
0xC0000904	Error	TDX_CPUID_LEAF_1F_FORMAT_UNRECOGNIZED	0	CPUID leaf 1F format is not recognized or sub-leaves are not in order.
0xC0000905	Error	TDX_INVALID_WBINVD_SCOPE	0	WBINVD scope is not supported.
0xC0000906	Error	TDX_INVALID_PKG_ID	Package ID	Package ID is larger than the maximum supported.
0xC0000908	Error	TDX_CPUID_LEAF_NOT_SUPPORTED	CPUID leaf	
0xC0000910	Error	TDX_SMRR_NOT_LOCKED	0: SMRR, 1: SMRR2	SMRR* is not locked.
0xC0000911	Error	TDX_INVALID_SMRR_CONFIGURATION	0: SMRR, 1: SMRR2	SMRR* configuration is invalid.
0xC0000912	Error	TDX_SMRR_OVERLAPS_CMR	Bits 7:0: 0: SMRR, 1: SMRR2 Bits 15:8: Overlapping CMR index	SMRR* overlaps a CMR.
0xC0000913	Error	TDX_SMRR_LOCK_NOT_SUPPORTED	0	Platform does not support SMRR locking.
0xC0000914	Error	TDX_SMRR_NOT_SUPPORTED	0	Platform does not support SMRR.
0xC0000920	Error	TDX_INCONSISTENT_MSR	Bits 31:0: MSR index	MSR configuration is inconsistent between LPs.
0xC0000921	Error	TDX_INCORRECT_MSR_VALUE	Bits 31:0: MSR index	MSR value is incorrect.
0xC0000930	Error	TDX_SEAMREPORT_NOT_AVAILABLE	0	SEAMOPS(SEAMREPORT) instruction leaf is not available.
0xC0000A00	Error	TDX_INVALID_TDMR	Bits 7:0: TDMR index	TDMR base address is not aligned on 1GB, its HKID bits are not 0, TDMR size is not specified with 1GB granularity or TDMR is outside the platform's maximum PA.
0xC0000A01	Error	TDX_NON_ORDERED_TDMR	Bits 7:0: TDMR index	TDMR is not specified in an ascending, non-overlapping order.
0xC0000A02	Error	TDX_TDMR_OUTSIDE_CMRS	Bits 7:0: TDMR index	TDMR non-reserved parts are not fully contained in CMRs.
0x00000A03	Success	TDX_TDMR_ALREADY_INITIALIZED	0	TDMR is already fully initialized.
0xC0000A10	Error	TDX_INVALID_PAMT	Bits 7:0: TDMR index Bits 15:8: PAMT level (2: 1GB, 1: 2MB, 0: 4KB)	PAMT region base address is not aligned on 4KB, its HKID bits are not 0, PAMT region size is not specified with 4KB granularity, it is not large enough for the TDMR size or PAMT region is outside the platform's maximum PA.
0xC0000A11	Error	TDX_PAMT_OUTSIDE_CMRS	Bits 7:0: TDMR index Bits 15:8: PAMT level (2: 1GB, 1: 2MB, 0: 4KB)	PAMT is not fully contained in CMRs.
0xC0000A12	Error	TDX_PAMT_OVERLAP	Bits 7:0: TDMR index Bits 15:8: PAMT level (2: 1GB, 1: 2MB, 0: 4KB) Bits 23:16: Overlapping TDMR index	PAMT overlaps with TDMR non-reserved parts or with another PAMT.
0xC0000A20	Error	TDX_INVALID_RESERVED_IN_TDMR	Bits 7:0: TDMR index Bits 15:8: Reserved area index	Reserved area in TMDR's base offset is not aligned on 4KB, its size is not specified with 4KB granularity or it is not fully contained within the TDMR.
0xC0000A21	Error	TDX_NON_ORDERED_RESERVED_IN_TDMR	Bits 7:0: TDMR index Bits 15:8: Reserved area index	Reserved area in TDMR is not specified in an ascending, non-overlapping order.
0xC0000A22	Error	TDX_CMR_LIST_INVALID	0	CMR list provided to the TDX module is invalid
0xC0000B00	Error	TDX_EPT_WALK_FAILED	Operand ID	EPT walk failed
0xC0000B01	Error	TDX_EPT_ENTRY_FREE	Operand ID	EPT entry is free
0xC0000B02	Error	TDX_EPT_ENTRY_NOT_FREE	Operand ID	EPT entry is not free
0xC0000B03	Error	TDX_EPT_ENTRY_NOT_PRESENT	Operand ID	EPT entry is not present
0xC0000B04	Error	TDX_EPT_ENTRY_NOT_LEAF	Operand ID	EPT entry is not a leaf
0xC0000B05	Error	TDX_EPT_ENTRY_LEAF	Operand ID	EPT entry is a leaf
0xC0000B06	Error	TDX_GPA_RANGE_NOT_BLOCKED	Operand ID	GPA range is not blocked
0x00000B07	Success	TDX_GPA_RANGE_ALREADY_BLOCKED	Operand ID	GPA range is already blocked
0xC0000B08	Error	TDX_TLB_TRACKING_NOT_DONE	Operand ID	TLB tracking has not been done
0xC0000B09	Error	TDX_EPT_INVALID_PROMOTE_CONDITIONS	Operand ID	Conditions for GPA mapping promotions as invalid
0x00000B0A	Success	TDX_PAGE_ALREADY_ACCEPTED	Error EPT level	Page has already been accepted
0xC0000B0B	Error	TDX_PAGE_SIZE_MISMATCH	Error EPT level	Requested page size does not match the current GPA mapping size

19.1.3. Function Completion Status Operand IDs

Table 19.3: Function Completion Operand IDs Definition

Operand ID	Explicit/ Implicit	Class	Operand	Description
0	Explicit	GPR	RAX	Explicit input operand RAX
1	Explicit	GPR	RCX	Explicit input operand RCX
2	Explicit	GPR	RDX	Explicit input operand RDX
3	Explicit	GPR	RBX	Explicit input operand RBX
4	Explicit	GPR	Reserved_RSP	Reserved
5	Explicit	GPR	RBP	Explicit input operand RBP
6	Explicit	GPR	RSI	Explicit input operand RSI
7	Explicit	GPR	RDI	Explicit input operand RDI
8	Explicit	GPR	R8	Explicit input operand R8
9	Explicit	GPR	R9	Explicit input operand R9
10	Explicit	GPR	R10	Explicit input operand R10
11	Explicit	GPR	R11	Explicit input operand R11
12	Explicit	GPR	R12	Explicit input operand R12
13	Explicit	GPR	R13	Explicit input operand R13
14	Explicit	GPR	R14	Explicit input operand R14
15	Explicit	GPR	R15	Explicit input operand R15
63-16	Explicit	Reserved	Reserved	Reserved for additional explicit operands (e.g., XMM).
64	Explicit	Component of explicit input	ATTRIBUTES	TD_PARAMS.ATTRIBUTES
65	Explicit	Component of explicit input	XFAM	TD_PARAMS.XFAM
66	Explicit	Component of explicit input	EXEC_CONTROLS	TD_PARAMS.EXEC_CONTROLS
67	Explicit	Component of explicit input	EPTP_CONTROLS	TD_PARAMS.EPTP_CONTROLS
68	Explicit	Component of explicit input	MAX_VCPUS	TD_PARAMS.MAX_VCPUS
69	Explicit	Component of explicit input	CPUID_CONFIG	TD_PARAMS.CPUID_CONFIG
70	Explicit	Component of explicit input	TSC_FREQUENCY	TD_PARAMS.TSC_FREQUENCY
95-71	Explicit	Component of explicit input	Reserved	Reserved for additional components.
96	Explicit	Component of explicit input	TDMR_INFO_PA	TDMR_INFO_PA array entry
127-97	Explicit	Component of explicit input	Reserved	Reserved for additional components.
128	Implicit	Physical Page	TDR	TDR Page
129	Implicit	Physical Page	TDCX	TDCX Page
130	Implicit	Physical Page	TDVPR	TDVPR Page
131	Implicit	Physical Page	TDVPX	TDVPX Page
143-132	Implicit	Physical Page	Reserved	Reserved for additional implicit physical page types.
144	Implicit	Logical control structure	TDCS	TDCS Logical Structure

Operand ID	Explicit/ Implicit	Class	Operand	Description
145	Implicit	Logical control structure	TDVPS	TDVPS Logical Structure
146	Implicit	Logical control structure	SEPT	Secure EPT Tree
167-147	Implicit	Logical control structure	Reserved	Reserved for additional implicit logical structure types.
168	Implicit	Component of logical control structure	RTMR	TDCS.RTMR
169	Implicit	Component of logical control structure	TD_EPOCH	TDCS.TD_EPOCH
183-170	Implicit	Component of logical control structure	Reserved	Reserved for additional components.
184	Implicit	Abstract item	SYS	Intel TDX Module
185	Implicit	Abstract item	TDMR	TDMR
186	Implicit	Abstract item	KOT	KOT
187	Implicit	Abstract item	KET	KET
188	Implicit	Abstract item	WBCACHE	TDH.PHYMEM.CACHE.WB State

20.ABI Reference: Data Types

This section describes data types that are designed to be used by the Intel TDX module.

20.1.1. Mutex

A mutex protects resources that must be accessed in an exclusive mode by a single LP. Architecturally, a mutex is implemented as a state variable (may be a single bit), as described below.

Table 20.1: Mutex State

State	Description
Free	No logical processor currently has an exclusive access to the resource protected by this lock
Exclusive	One logical processor has an exclusive access to the resource protected by this lock

Mutexes implement the methods described below. The following points should be noted:

- There is **no notion of waiting on a lock**. The operation either succeeds or fails immediately.
- The description below assume that the state transition operation occurs **atomically**.
- Implementation may add defense-in-depth for illegal conditions, e.g., attempt to Release() a mutex by an LP that hasn't acquired that mutex.

Table 20.2: Mutex Methods

Method	Previous State	Next State	Result
Acquire()	Free	Exclusive	Success
	Exclusive	Exclusive	Fail
Release()	Free	N/A	Fatal error
	Exclusive	Free	Success

20.1.2. Shared/Exclusive (Readers/Writer) Lock

A shared/exclusive lock protects resources that can be accessed in either a shared mode by multiple LPs, or in an exclusive mode by a single LP. Architecturally, a shared lock is implemented as a state variable and a counter, as described below.

Note: The same lock type is sometime called **readers/writer lock**. That name is misleading since shared access may allow writing to the protected resource in some cases.

Table 20.3: Shared/Exclusive Lock State and Counter

State	LP Counter Value	Description
Free	0	No logical processor currently has an exclusive nor shared access to the resource protected by this lock
Shared	> 0	One or more logical processors have a shared access to the resource protected by this lock
Exclusive	0	One logical processor has an exclusive access to the resource protected by this lock

Shared/exclusive locks implement the methods described below. The following points should be noted:

- There is **no notion of waiting on a lock**. The operation either succeeds or fails immediately.
- The description below assume that the state transition and counter operation occurs **atomically**.

- Implementation may add defense-in-depth for illegal conditions, e.g., attempt to ReleaseExclusive() a lock by an LP that hasn't acquired that lock.

Table 20.4: Shared/Exclusive Lock Methods

Method	Description	Previous State	Previous LP Counter	Next State	LP Counter Operation	Result
AcquireExclusive()	Acquire an exclusive access	Free	0	Exclusive	None	Success
		Shared	≥ 1	Shared	None	Fail
		Exclusive	0	Exclusive	None	Fail
AcquireShared()	Acquire a shared access	Free	0	Shared	Increment	Success
		Shared	≥ 1	Shared	Increment	Success
		Exclusive	0	Exclusive	None	Fail
ReleaseExclusive()	Release a lock, previously acquired as exclusive by the current LP	Free	0	N/A	N/A	Fatal error
		Shared	≥ 1	N/A	N/A	Fatal error
		Exclusive	0	Free	None	Success
ReleaseShared()	Release a lock, previously acquired as shared by the current LP	Free	0	N/A	N/A	Fatal error
		Shared	≥ 1	Free	Decrement	Success
		Exclusive	0	N/A	N/A	Fatal error
Promote()	Reacquire a lock, previously acquired as shared by the current LP, as exclusive	Free	0	N/A	N/A	Fatal error
		Shared	1	Exclusive	Clear to 0	Success
			> 1	Shared	None	Fail
		Exclusive	N/A	N/A	N/A	Fatal error
Demote() **(Currently not in Use)**	Reacquire a lock, previously acquired as exclusive by the current LP, as shared	Free	0	N/A	N/A	Fatal error
		Shared	≥ 1	N/A	N/A	Fatal error
		Exclusive	0	Shared	Set to 1	Success

Implementation Notes

- The counter should be implemented with enough bits to account for the maximum number of times the lock may be concurrently acquired. A single TDX function may have several operands, and a misbehaving software may provide the same address for all of them. This may lead to a single function acquiring the same lock multiple times (harmlessly releasing them later as the error condition is detected). Thus, if N is the maximum number of operands per function and MAX_LPS is the maximum number of logical processors in the platform, the maximum value of the lock's counter is N * MAX_LPS.

20.2. Basic Crypto Types

Table 20.5: Basic Crypto Types

Name	Size (Bytes)	Description
SHA384_HASH	48	384-bit buffer containing the result of a SHA384 hash calculation
KEY128	16	128-bit key
KEY256	32	256-bit key

20.3. TD Parameters Types

Note: This section describes TD parameter types, as defined. Implementation details may differ.

20.3.1. ATTRIBUTES

ATTRIBUTES is defined as a 64b field that specifies various guest TD attributes. ATTRIBUTES is provided by the host VMM as a guest TD initialization parameter as part of TD_PARAMS. It is reported to the guest TD by TDG.VP.INFO and as part of TDREPORT_STRUCT returned by TDG.MR.REPORT.

The ATTRIBUTES bits are divided into three groups, as shown in the table below. If any bit in the TUD group is set to 1, the guest TD is under off-TD debug and is untrusted. The SEC group indicates features that may impact TD security but are not considered as impacting TD trust.

Table 20.6: ATTRIBUTES Definition

Bits	Group	Description	Bits	Field	Description	Reference
7:0	**TUD**	TD Under Debug If any of the bits in this group are set to 1, the guest TD is untrusted.	0	**DEBUG**	Guest TD runs in off-TD debug mode. Its VCPU state and private memory are accessible by the host VMM.	14.3
			7:1	**RESERVED**	Reserved for future TUD flags – must be 0	
31:8	**SEC**	Attributes that may impact TD security	27:8	**RESERVED**	Reserved for future SEC flags – must be 0	
			28	**SEPT_VE_DISABLE**	Disable EPT violation conversion to #VE on guest TD access of PENDING pages	
			29	**RESERVED**	Reserved for future SEC flags – must be 0	
			30	**PKS**	TD is allowed to use Supervisor Protection Keys.	10.14
			31	**KL**	TD is allowed to use Key Locker. Must be 0	
63:32	**OTHER**		62:32	**RESERVED**	Reserved for future OTHER flags – must be 0	

		Attributes that do not impact TD security	63	**PERFMON**	TD is allowed to use Perfmon and PERF_METRICS capabilities.	14.2

20.3.2. XFAM

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

Intel TDX module extended state handling is described in 10.5.

XFAM (eXtended Features Available Mask) is defined as a 64b bitmap, which has the same format as XCR0 or IA32_XSS MSR. XFAM determines the set of extended features available for use by the guest TD. XFAM is provided by the host VMM as a guest TD initialization parameter as part of TD_PARAMS. It is reported to the guest TD by CPUID(0x0D, 0x01) and as part of TDREPORT_STRUCT returned by TDG.MR.REPORT.

The Intel TDX module and the Intel® architecture impose some rules on how the bits of XFAM may be set. See Table 10.4 for details.

20.3.3. CPUID_VALUES

CPUID_VALUES is defined as a 128b structure composed of four 32b fields representing the values returned by CPUID in registers EAX, EBX, ECX and EDX. An array of CPUID_RET is used during guest TD configuration by TDH.MNG.INIT.

Table 20.7: CPUID_VALUES Definition

Field	Offset (Bytes)	Size (Bytes)	Description
EAX	0	4	Value returned by CPUID in EAX
EBX	4	4	Value returned by CPUID in EBX
ECX	8	4	Value returned by CPUID in ECX
EDX	12	4	Value returned by CPUID in EDX

20.3.4. TD_PARAMS

TD_PARAMS is provided as an input to TDH.MNG.INIT, and some of its fields are included in the TD report. The format of this structure is valid for a specific MAJOR_VERSION of the Intel TDX module, as reported by TDH.SYS.INFO.

TD_PARAMS' size is 1024B.

Table 20.8: TD_PARAMS Definition

Field	Offset (Bytes)	Type	Size (Bytes)	Description	Included in TDREPORT?
ATTRIBUTES	0	64b bitmap (see 20.3.1)	8	TD attributes: the value set in this field must comply with ATTRIBUTES_FIXED0 and ATTRIBUTES_FIXED1 enumerated by TDH.SYS.INFO.	Yes

Field	Offset (Bytes)	Type	Size (Bytes)	Description	Included in TDREPORT?
XFAM	8	64b bitmap in XCR0 format	8	Extended Features Available Mask: indicates the extended state features allowed for the TD. XFAM's format is the same as XCR0 and IA32_XSS MSR. The value set in this field must satisfy the following conditions: • Natively valid value for XCR0 and IA32_XSS (does not contain reserved bits, features not supported by the CPU, or illegal bit combinations) • Complies with XFAM_FIXED0 and XFAM_FIXED1 as enumerated by TDH.SYS.INFO.	Yes
MAX_VCPUS	16	Unsigned 16b Integer	2	Maximum number of VCPUs	No
RESERVED	18	N/A	6	Must be 0	No
EPTP_CONTROLS	24	EPTP	8	Control bits of EPTP – copied to each TD VMCS on TDH.VP.INIT: **Bits 2:0** Memory type – must be 110 (WB) **Bits 5:3** EPT level – 1 less than the EPT page-walk length **Bits 63:6** Reserved – must be 0	No
EXEC_CONTROLS	32	64b bitmap (see Table 20.9)	8	Non-measured TD-scope execution controls	No
TSC_FREQUENCY	40	16b unsigned integer	2	TD-scope virtual TSC frequency in units of 25MHz – must be between 4 and 400.	No
RESERVED	42	N/A	38	Must be 0	No
MRCONFIGID	80	SHA384_HASH	48	Software-defined ID for non-owner-defined configuration of the guest TD – e.g., run-time or OS configuration	Yes
MROWNER	128	SHA384_HASH	48	Software-defined ID for the guest TD's owner	Yes
MROWNERCONFIG	176	SHA384_HASH	48	Software-defined ID for owner-defined configuration of the guest TD – e.g., specific to the workload rather than the run-time or OS	Yes
RESERVED	224	N/A	32	Must be 0	No
CPUID_CONFIG[0]	256	CPUID_VALUES	16		No

Field	Offset (Bytes)	Type	Size (Bytes)	Description	Included in TDREPORT?
				Direct configuration of CPUID leaves/sub-leaves virtualization: the number and order of entries must be equal to the number and order of directly configurable or allowable CPUID leaves/sub-leaves reported by TDH.SYS.INFO. Note that the leaf and sub-leaf numbers are implicit. Only bits that have been reported as 1 by TDH.SYS.INFO may be set to 1. Note that the virtualization of many CPUID bit fields not enumerated in this list is configurable indirectly, via the XFAM and ATTRIBUTES fields.	
CPUID_CONFIG[n-1]		CPUID_VALUES	16		
RESERVED		N/A		Fills up to TD_PARAMS size (1024B) – must be 0	No

Table 20.9: TD_PARAMS_STRUCT.EXEC_CONTROLS Definition

Bits	Name	Description
0	**GPAW**	TD-scope Guest Physical Address Width execution control: copied to each TD VMCS **GPAW** execution control on TDH.VP.INIT 0: GPA.SHARED bit is GPA[47] 1: GPA.SHARED bit is GPA[51]
63:1	**RESERVED**	Must be 0

20.4. Physical Memory Management Types

Note: This section describes physical memory types, as defined. Implementation may differ.

PAMT entry and PT (page type) are defined in 7.3.

20.4.1. Physical Page Size

Three physical page size levels (4KB, 2MB and 1GB) are defined.

Table 20.10: Page Size Definition

Page Size	Associated Physical Page Size	Value
PS_1G	1GB	2
PS_2M	2MB	1
PS_4K	4KB	0

20.5. TD Private Memory Management Data Types: Secure EPT

Intel SDM, Vol. 3, 28.2.2 EPT Translation Mechanism

Note: This section describes private memory management types, as defined. Implementation may differ.

20.5.1. Secure EPT Levels

Secure EPT level definition is identical to legacy VMX EPT level definition. As a rule, an EPT entry at level L maps a GPA range whose size is 2^{12+9*L}.

Table 20.11: EPT Levels Definition

Level	0	1	2	3	4	5 (5-Level EPT Only)
GPA Range Size	4KB	2MB	1GB	512GB	256TB	16PB[7]
Child Physical Page Size	4KB	2MB	1GB	N/A	N/A	N/A
EPT Page Type	N/A	EPT	EPD	EPDPT	EPML4	EPML5
Parent EPT Entry Type	EPTE	EPDE	EPDPTE	EPML4E	EPML5E (5-level EPT) or VMCS.EPTP (4-level EPT)	VMCS.EPTP
GPA Offset Bits	20:12	29:21	38:30	47:39	51:48 (5-level EPT only)	N/A

20.5.2. Secure EPT Entry Information as Returned by TDX Module Functions

Many Intel TDX module functions return Secure EPT entry information. This information is returned in the formats detailed below, which may be different that the actual Secure EPT format as maintained by the TDX module in memory.

Note: The returned Secure EPT information is subject to change with new versions of TDX.

20.5.2.1. *Returned Secure EPT Entry Content*

The returned secure EPT entry format is detailed below. It may be different that the actual Secure EPT format as maintained by the TDX module in memory.

Table 20.12: Secure EPT Entry Content as Returned by TDX Interface Functions

Secure EPT Entry Field						Value Returned in RCX (per Entry State Returned in RDX)		
MSB	**LSB**	**Size**	**Short Name**	**Full Name**	**Enabled**	**Leaf**	**Non-Leaf**	**SEPT_FREE**
0	0	1	R	Read	N/A	R	R	0
1	1	1	W	Write	N/A	W	W	0
2	2	1	X	Execute	N/A	X	X	0
5	3	3	MT	Memory Type	N/A	6	0	0
6	6	1	IPAT	Ignore PAT	N/A	1	0	0
7	7	1	PS	Leaf	N/A	1	0	0
8	8	1	A	Accessed	No	0	0	0
9	9	1	D	Dirty	No	0	0	0
10	10	1	Xu	Execute (User)	No	0	0	0
11	11	1	Ignored	Ignored	N/A	0	0	0
51	12	40	HPA[51:12]	Host Physical Address [51:12]	N/A	HPA[51:12]	HPA[51:12]	0
57	57	1	VPW	Verify Paging-Write	No	0	0	0
58	58	1	PW	Paging Write	No	0	0	0

[7] Only the lower half is available as TD private GPA space, because the SHARED bit must be 0

Secure EPT Entry Field						Value Returned in RCX (per Entry State Returned in RDX)		
MSB	LSB	Size	Short Name	Full Name	Enabled	Leaf	Non-Leaf	SEPT_FREE
59	59	1	Ignored	Ignored	N/A	0	0	0
60	60	1	SSS	Supervisor Shadow Stack	No	0	0	0
61	61	1	SPP	Check Sub-Page Permissions	No	0	0	0
62	62	1	Ignored	Ignored	N/A	0	0	0
63	63	1	SVE	Suppress #VE	Yes	SVE	0	1

20.5.2.2. *Additional Returned Secure EPT Information*

Additional information for secure EPT entries is returned as defined below.

Table 20.13: Additional Secure EPT Entry Information Returned by TDX Interface Functions

Bits	Name	Description
2:0	Level	Level of the returned Secure EPT entry – see 20.5.1 above
7:3	Reserved	Set to 0
15:8	State	The TDX state of the Secure EPT entry – see Table 20.14 below
63:16	Reserved	Set to 0

Table 20.14: Secure EPT Entry TDX State Returned by TDX Interface Functions

Value	Secure EPT Entry State
0	SEPT_FREE
1	SEPT_BLOCKED
2	SEPT_PENDING
3	SEPT_PENDING_BLOCKED
4	SEPT_PRESENT
Other	Reserved

20.6. TD Entry and Exit Types

20.6.1. Extended Exit Qualification

Extended Exit Qualification is a 64-bit field returned by TDH.VP.ENTER for asynchronous TD exits with an architectural VMX exit reasons. It contains additional non-VMX, TDX-specific information.

Table 20.15: Extended Exit Qualification

Bits	Name	Description		
3:0	TYPE	Extended exit qualification type		
		Value	**Name**	**Description**
		0	NONE	No extended exit qualification
		1	ACCEPT	Exit qualification for an EPT violation during TDG.MEM.PAGE.ACCEPT
		Other	Reserved	
31:4	Reserved	Set to 0		
63:32	INFO	TYPE-specific information Set to 0 for NONE – See below for other values of TYPE		

Table 20.16: Extended Exit Qualification INFO Field for ACCEPT

Bits	Name	Description
34:32	REQ_SEPT_LEVEL	SEPT level requested as an input to TDG.MEM.PAGE.ACCEPT
37:35	ERR_SEPT_LEVEL	SEPT level in which TDG.MEM.PAGE.ACCEPT detected an error
45:38	ERR_SEPT_STATE	The TDX state of the Secure EPT entry where TDG.MEM.PAGE.ACCEPT detected an error – see Table 20.14 above
46	ERR_SEPT_IS_LEAF	Indicates that the SEPT entry where TDG.MEM.PAGE.ACCEPT detected an error is a leaf entry
63:47	Reserved	Set to 0

20.7. Measurement and Attestation Types

Note: This section describes measurement and attestation types, as defined. Implementation may differ.

20.7.1. CPUSVN

CPUSVN is a 16B Security Version Number of the CPU.

- There is a single CPUSVN used for SGX and TDX.
- CPUSVN contents are considered micro-architectural. CPUSVN is composed of fields for PR_RESET_SVN, R_LAST_PATCH_SVN, SINIT, BIOS ACM, Boot Guard ACM and BIOS Guard NP-PPPE module.

20.7.2. TDREPORT_STRUCT

TDREPORT_STRUCT is the output of the TDG.MR.REPORT function. TDREPORT_STRUCT is composed of a generic MAC structure (REPORTMACSTRUCT, see 20.7.3 below), a SEAMINFO structure and a TDX-specific TEE info structure (TDINFO_STRUCT, see 20.7.5 below).

The size of TDREPORT_STRUCT is 1024B.

Table 20.17: TDREPORT_STRUCT Definition

Name	Offset (Bytes)	Type	Size (Bytes)	Description
REPORTMACSTRUCT	0	REPORTMACSTRUCT	256	REPORTMACSTRUCT for the TDG.MR.REPORT
TEE_TCB_INFO	256	TEE_TCB_INFO_STRUCT	239	Additional attestable elements in the TD's TCB are not reflected in the REPORTMACSTRUCT.CPUSVN – includes the Intel TDX module measurements.
RESERVED	495	N/A	17	Reserved – contains 0
TDINFO	512	TDINFO_STRUCT	512	TD's attestable properties

20.7.3. REPORTMACSTRUCT (Reference)

REPORTMACSTRUCT is common to Intel's Trusted Execution Environments (TEEs) – e.g., SGX and TDX. REPORTMACSTRUCT is the first field in the TEE report structure. In the TDX architecture, that is TDREPORT_STRUCT. REPORTMACSTRUCT is MAC-protected and contains hashes of the remainder of the report structure which includes the TEE's measurements, and where applicable, the measurements of additional TCB elements not reflected in REPORTMACSTRUCT.CPUSVN – e.g., a SEAM's measurements.

Software verifying a TEE report structure (for TDX, this includes TEE_TCB_INFO_STRUCT and TDINFO_STRUCT) should first confirm that its REPORTMACSTRUCT.TEE_TCB_INFO_HASH equals the hash of the TEE_TCB_INFO_STRUCT (if applicable) and that REPORTMACSTRUCT.TEE_INFO_HASH equals the hash of the TDINFO_STRUCT. Then, software uses ENCLU(EVERIFYREPORT) to help check the integrity of the REPORTMACSTRUCT. If all checks pass, the measurements in the structure describe a TEE on this platform.

The size of REPORTMACSTRUCT is 256B.

Table 20.18: REPORTMACSTRUCT Definition

Name	Offset (Bytes)	Type	Size (Bytes)	Description	MAC
REPORTTYPE	0	REPORTTYPE	4	Type Header Structure	Yes
RESERVED	4		12	Must be zero	Yes
CPUSVN	16	CPUSVN	16	CPU SVN	Yes
TEE_TCB_INFO_HASH	32	SHA384_HASH	48	SHA384 of TEE_TCB_INFO for TEEs implemented using Intel TDX	Yes
TEE_INFO_HASH	80	SHA384_HASH	48	SHA384 of TEE_INFO: a TEE-specific info structure (TDINFO_STRUCT or SGXINFO) or 0 if no TEE is represented	Yes
REPORTDATA	128		64	A set of data used for communication between the caller and the target.	Yes
RESERVED	192		32	Must be zero	Yes
MAC	224		32	The MAC over the REPORTMACSTRUCT with model-specific MAC	No

20.7.4. REPORTTYPE (Reference)

REPORTTYPE indicates the reported Trusted Execution Environment (TEE) type, sub-type and version.

The size of REPORTTYPE is 4B.

Table 20.19: REPORTTYPE Definition

Name	Offset (Bytes)	Size (Bytes)	Description
TYPE	0	1	Trusted Execution Environment (TEE) Type: 0x00: SGX 0x7F-0x01: Reserved (TEE implemented by CPU) 0x80: Reserved (TEE implemented by SEAM module) 0x81: TDX 0xFF-0x82: Reserved (TEE implemented by SEAM module)
SUBTYPE	1	1	TYPE-specific subtype Value is 0
VERSION	2	1	TYPE-specific version. Value is 0
RESERVED	3	1	Must be zero

20.7.5. TDINFO_STRUCT

TDINFO_STRUCT is defined as the TDX-specific TEE_INFO part of TDG.MR.REPORT. It contains the measurements and initial configuration of the TD that was locked at initialization and a set of measurement registers that are run-time extendable. These values are copied from the TDCS by the TDG.MR.REPORT function. Refer to 10.12 for additional details.

The size of TDINFO_STRUCT is 512B.

Table 20.20: TDINFO_STRUCT Definition

Name	Offset (Bytes)	Type	Size (Bytes)	Description
ATTRIBUTES	0		8	TD's ATTRIBUTES
XFAM	8		8	TD's XFAM
MRTD	16	SHA384_HASH	48	Measurement of the initial contents of the TD
MRCONFIGID	64	SHA384_HASH	48	Software-defined ID for non-owner-defined configuration of the guest TD – e.g., run-time or OS configuration
MROWNER	112	SHA384_HASH	48	Software-defined ID for the guest TD's owner
MROWNERCONFIG	160	SHA384_HASH	48	Software-defined ID for owner-defined configuration of the guest TD – e.g., specific to the workload rather than the run-time or OS
RTMR	208	SHA384_HASH	NUM_RTMRS * 48	Array of NUM_RTMRS (4) run-time extendable measurement registers
RESERVED	400	N/A	112	Must be zero

20.8. Configuration, Enumeration and Initialization Types

Note: This section describes configuration, enumeration and initialization types, as defined. Implementation may differ.

20.8.1. CPUID_CONFIG

CPUID_CONFIG is designed to enumerate how the host VMM may configure the virtualization done by the Intel TDX module for a single CPUID leaf and sub-leaf. An array of CPUID_CONFIG entries is used for the Intel TDX module enumeration by TDH.SYS.INFO.

Table 20.21: CPUID_CONFIG Definition

Field	Offset (Bytes)	Size (Bytes)	Description
LEAF	0	4	EAX input value to CPUID
SUB_LEAF	4	4	ECX input value to CPUID A value of -1 indicates a CPUID leaf with no sub-leaves.
EAX	8	4	Enumeration of the configurable virtualization of the value returned by CPUID in EAX: a value of 1 in any of the bits indicates that the host VMM is allowed to configure that bit
EBX	12	4	Enumeration of the configurable virtualization of the value returned by CPUID in EBX: a value of 1 in any of the bits indicates that the host VMM is allowed to configure that bit
ECX	16	4	Enumeration of the configurable virtualization of the value returned by CPUID in ECX: a value of 1 in any of the bits indicates that the host VMM is allowed to configure that bit
EDX	20	4	Enumeration of the configurable virtualization of the value returned by CPUID in EDX: a value of 1 in any of the bits indicates that the host VMM is allowed to configure that bit

20.8.2. TDSYSINFO_STRUCT

TDSYSINFO_STRUCT is designed to provide enumeration information about the Intel TDX module. It is an output of the TDH.SYS.INFO leaf function.

TDSYSINFO_STRUCT's size is 1024B.

Table 20.22: TDSYSINFO_STRUCT Definition

Section	Field Name	Offset (Bytes)	Type	Size (Bytes)	Description
Intel TDX Module Info	**ATTRIBUTES**	0	Bitmap	4	Module attributes **Bits 30:0** Reserved – set to 0 **Bit 31** 0 indicates a production module. 1 indicates a debug module.
	VENDOR_ID	4	Integer	4	0x8086 for Intel

Section	Field Name	Offset (Bytes)	Type	Size (Bytes)	Description
	BUILD_DATE	8	BCD	4	Intel TDX module build data – in yyyymmdd BCD format (each digit occupies 4 bits)
	BUILD_NUM	12	Integer	2	Build number of the Intel TDX module
	MINOR_VERSION	14	Integer	2	Minor version number of the Intel TDX module
	MAJOR_VERSION	16	Integer	2	Major version number of the Intel TDX module
	RESERVED	18	N/A	14	This field is reserved for enumerating future Intel TDX module capabilities. Set to 0
Memory Info	**MAX_TDMRS**	32	Integer	2	The maximum number of TDMRs supported
	MAX_RESERVED_ PER_TDMR	34	Integer	2	The maximum number of reserved areas per TDMR
	PAMT_ENTRY_ SIZE	36	Integer	2	The size of a PAMT entry – determines the number of bytes that need to be reserved for the three PAMT areas: • PAMT_1G (1 entry per 1GB of TDMR) • PAMT_2M (1 entry per 2MB of TDMR) • PAMT_4K (1 entry per 4KB of TDMR)
	RESERVED	38	N/A	10	Set to 0
Control Struct Info	**TDCS_BASE_SIZE**	48	Integer	2	Base value for the number of bytes required to hold TDCS
	RESERVED	50	N/A	2	Reserved for additional TDCS enumeration Set to 0
	TDVPS_BASE_SIZE	52	Integer	2	Base value for the number of bytes required to hold TDVPS
	RESERVED	54	N/A	10	Set to 0

Section	Field Name	Offset (Bytes)	Type	Size (Bytes)	Description
TD Capabilities	**ATTRIBUTES_ FIXED0**	64	Bitmap	8	If any certain bit is 0 in ATTRIBUTES_FIXED0, it must be 0 in any TD's ATTRIBUTES. The value of this field reflects the Intel TDX module capabilities and configuration and CPU capabilities.
	ATTRIBUTES_ FIXED1	72	Bitmap	8	If any certain bit is 1 in ATTRIBUTES_FIXED1, it must be 1 in any TD's ATTRIBUTES. The value of this field reflects the Intel TDX module capabilities and configuration and CPU capabilities.
	XFAM_FIXED0	80	Bitmap	8	If any certain bit is 0 in XFAM_FIXED0, it must be 0 in any TD's XFAM.
	XFAM_FIXED1	88	Bitmap	8	If any certain bit is 1 in XFAM_FIXED1, it must be 1 in any TD's XFAM.
	RESERVED	96	N/A	32	Set to 0
	NUM_CPUID_ CONFIG	128	Integer	4	Number of the following CPUID_CONFIG entries
	CPUID_CONFIG[0]	132	CPUID_ CONFIG	24	Enumeration of the CPUID leaves/sub-leaves that contain bit fields whose virtualization by the Intel TDX module is either: • Directly configurable (CONFIG_DIRECT) by the host VMM • Bits that the host VMM may allow to be 1 (ALLOW_DIRECT) and their native value, as returned by the CPU, is 1. See 20.8.1 for details. Note that the virtualization of many CPUID bit fields not enumerated in this list is configurable indirectly via the XFAM and ATTRIBUTES assigned to a TD by the host VMM.
	CPUID_ CONFIG[last]		CPUID_ CONFIG	24	
Reserved	**RESERVED**		N/A		Fills up to the structure size (1024B) – set to 0

20.8.3. CMR_INFO

CMR_INFO is designed to provide information about a Convertible Memory Range (CMR), as configured by BIOS and checked and stored securely by MCHECK.

Table 20.23: CMR_INFO Entry Definition

Name	Offset (Bytes)	Type	Size (Bytes)	Description
CMR_BASE	0	Physical Address	8	Base address of the CMR: since a CMR is aligned on 4KB, bits 11:0 are 0.
CMR_SIZE	8	Integer	8	Size of the CMR, in bytes: since a CMR is aligned on 4KB, bits 11:0 are 0. A value of 0 indicates a null entry.

TDH.SYS.INFO leaf function returns an array of CMR_INFO entries. The CMRs are sorted from the lowest base address to the highest base address, and they are non-overlapping.

20.8.4. TDMR_INFO

TDMR_INFO is designed to provide information about a single Trust Domain Memory Region (TDMR) and its associated PAMT.

Table 20.24: TDMR_INFO Entry Definition

Name	Offset (Bytes)	Type	Size (Bytes)	Description
TDMR_BASE	0	Physical Address	8	Base address of the TDMR (HKID bits must be 0): since a TDMR is aligned on 1GB, bits 29:0 are always 0.
TDMR_SIZE	8	Integer	8	Size of the TDMR, in bytes: must be greater than 0 and a whole multiple of 1GB (i.e., bits 29:0 are always 0).
PAMT_1G_BASE	16	Physical Address	8	Base address of the PAMT_1G range associated with the above TDMR (HKID bits must be 0): since a PAMT range is aligned on 4KB, bits 11:0 are always 0.
PAMT_1G_SIZE	24	Integer	8	Size of the PAMT_1G range associated with the above TDMR: since a PAMT range is aligned on 4KB, bits 11:0 are always 0.
PAMT_2M_BASE	32	Physical Address	8	Base address of the PAMT_2M range associated with the above TDMR (HKID bits must be 0): since a PAMT range is aligned on 4KB, bits 11:0 are always 0.
PAMT_2M_SIZE	40	Integer	8	Size of the PAMT_2M range associated with the above TDMR: since a PAMT range is aligned on 4KB, bits 11:0 are always 0.
PAMT_4K_BASE	48	Physical Address	8	Base address of the PAMT_4K range associated with the above TDMR (HKID bits must be 0): since a PAMT range is aligned on 4KB, bits 11:0 are always 0.
PAMT_4K_SIZE	56	Integer	8	Size of the PAMT_4K range associated with the above TDMR: since a PAMT range is aligned on 4KB, bits 11:0 are always 0.
RESERVED_OFFSET[0]	64	Integer	8	• Offset of reserved range 0 within the TDMR: since a reserved range is aligned on 4KB, bits 11:0 are always 0.
RESERVED_SIZE[0]	72	Integer	8	Size of reserved range 0 within the TDMR:

Name	Offset (Bytes)	Type	Size (Bytes)	Description
				• A size of 0 indicates a null entry. All following reserved range entries must also be null. • Since a reserved range is aligned on 4KB, bits 11:0 are always 0.
RESERVED_OFFSET[N-1]	64 + 16*(N-1)	Integer	8	Offset of the last reserved range within the TDMR.
RESERVED_SIZE[N-1]	72 + 16*(N-1)	Integer	8	Size of the last reserved range within the TDMR.

Notes:

- The number of reserved areas within a TDMR is enumerated by TDSYSINFO_STRUCT.MAX_RESREVED_PER_TDMR (see 20.8.2).
- Within each TDMR entry, all reserved areas must be sorted from the lowest offset to the highest offset, and they must not overlap with each other.
- All TDMRs and PAMTs must be contained within CMRs.
- A PAMT area must not overlap with another PAMT area (associated with any TDMR), and it must not overlap with non-reserved areas of any TDMR. PAMT areas may reside within reserved areas of TDMRs.

20.9. Metadata Access Types

Note: This section describes control structure field access types, as defined. Implementation may differ.

Metadata access is described in 17.4.

20.9.1. MD_FIELD_ID: Metadata Field Identifier

A metadata field identifier is used for accessing a single metadata element. Lists of metadata field identifiers for TD-scope metadata and VCPU-scope metadata are provided in Ch. 21. To access a certain metadata element, a **base identifier** is taken from those tables.

Field access codes are used with TDH.MNG.RD, TDH.MNG.WR, TDH.VP.RD, TDH.VP.WR, TDG.VM.RD and TDG.VM.WR functions. They have the following general structure:

Table 20.25: Metadata Field Identifier Definition

Bits	Name	Description
63	NON_ARCH	A value of 0 indicates that the field code and field definition will be maintained throughout Intel TDX module updates. A value of 1 indicates that the field code and field definition are non-architectural, and their meaning may change with Intel TDX module version. This is normally used for fields accessible only in debug mode.
62:56	CLASS_CODE	Identifies the field class – see the following sections for details
55:32	RESERVED	Must be 0
31:0	FIELD_CODE	Identifies the field – see the following sections for details

The interface functions are designed to read and/or write up to eight bytes at a time. Thus, for those functions, fields that are larger than eight bytes are divided into multiple elements, each with its own field code. For example, SHA384 fields such as MRCONFIGID, whose size is 48B, are divided into six 8B elements, with six sequential field codes.

20.9.2. TDR and TDCS Metadata Fields

Intel SDM, Vol. 3, 24.6.9 MSR-Bitmap Address

TD-scope field identifiers are used with TDH.MNG.RD, TDH.MNG.WR, TDG.VM.RD and TDG.VM.WR.

TD-scope CLASS_CODE is defined as follows:

Table 20.26: TD Scope (TDR and TDCS) Metadata CLASS_CODE Definition

CLASS_CODE	Control Structure	Class Name	FIELD_CODE Meaning
0	TDR	TD Management	Arbitrary field identifiers
1	TDR	Key Management	Arbitrary field identifiers
16	TDCS	TD Management	Arbitrary field identifiers
17	TDCS	Execution Controls	Arbitrary field identifiers
18	TDCS	TLB Epoch Tracking	Arbitrary field identifiers
19	TDCS	Measurement	Arbitrary field identifiers
32	TDCS	MSR Bitmaps	Offset (in 8B units) from the beginning of the architectural MSR bitmaps page structure, as specified by the [Intel SDM, Vol. 3, 24.6.9]
33	TDCS	Secure EPT Root	Offset (in 8B units) from the beginning of the page

20.9.3. TDVPS Metadata Field Codes

Intel SDM, Vol. 1, 13.4 XSAVE Area
Intel SDM, Vol. 3, 24.11.2 VMREAD, VMWRITE, and Encodings of VMCS Fields
Intel SDM, Vol. 3, 29.1 Virtual APIC State
Intel SDM, Vol. 3, App. B Field Encoding in VMCS

TDVPS field access codes are used with TDH.VP.RD and TDH.VP.WR, as described in 22.2.43 and 22.2.44, respectively.

TDVPS CLASS_CODE is defined as follows:

Table 20.27: TD VCPU Scope (TDVPS) Metadata CLASS_CODE Definition

CLASS_CODE	Class Name	FIELD_CODE Meaning
0	TD VMCS	Architectural VMCS field code, as specified by [Intel SDM, Vol. 3, 24.11.2 and App. B]. The “HIGH” access type (for accessing the upper 32b of 64b fields) is not supported.
1	VAPIC	Offset (in 8B units) from the beginning of the architectural virtual APIC page structure, as specified by the [Intel SDM, Vol. 3, 29.1]
2	VE_INFO	Arbitrary field identifiers
16	Guest GPR State	Architectural GPR number
17	Other Guest State	Arbitrary field identifiers
18	Guest Extended State	Offset (in 8B units) from the beginning of the architectural XSAVE area structure, as specified by the [Intel SDM, Vol. 1, 13.4] and enumerated by CPUID(0x0A)
19	Guest MSR State	Architectural MSR index, packed as shown below:

CLASS_CODE	Class Name	FIELD_CODE Meaning	
		Bits	**Description**
		31:14	Reserved, must be 0
		13	Bit 31 (equal to bit 30) of the architectural MSR index
		12:0	Bits 12:0 of the architectural MSR index
32	Management	Arbitrary field identifiers	

21.ABI Reference: Control Structures

This chapter describes the details of TDX control structures.

21.1. TD-Scope Control Structures

TD-scope control structures are described in 5.2.

21.1.1. How to Read the TDR and TDCS Tables

The VMM access column describes whether this field is accessible to the host VMM, using TDH.MNG.RD and TDDBGWRV, in production mode (ATTRIBUTES.DEBUG == 0) and debug mode (ATTRIBUTES.DEBUG == 1). Possible values are shown in the table below.

Table 21.1: VMM Access Definition

VMM Access	Meaning
None	No host VMM access to the field.
RO	Host VMM can only read the field using TDH.MNG.RD.
RW	Host VMM can read and write the field using TDH.MNG.RD and TDH.MNG.WR.

21.1.2. TDR

Note: This section describes TDR, as defined. Implementation may differ.

TDR is the root control structure of a guest TD. TDR is encrypted using the Intel TDX global private HKID. It contains the minimal set of fields that allow TD management operation when the guest TD's private ephemeral HKID is not known yet or when the TD's key state is such that memory encrypted with the guest TD's private ephemeral key is not accessible.

TDR occupies a single 4KB naturally aligned page of memory. It is the first TD page to be allocated and the last to be removed.

TRD fields are divided into the following classes:

Table 21.2: TDR Field Classes Definition

Field Class	Description
TD Management	These fields are used to manage the TDR page, its descendent TD private memory pages and control structure pages.
Key Management	These fields are used by the Intel TDX module to manage memory encryption keys. See Chapter 4 for details.

Note: The table below lists only TDR fields that may be accessed by the host VMM in either production or debug mode.

Table 21.3: TDR Definition

Class	Field	VMM Access		Type	Description	Field Code
		Prod.	Debug			
TD Management	INIT	None	RO	Boolean	Indicates that the TDCS has been initialized by TDH.MNG.INIT	0x8000000000000000
TD Management	FATAL	None	RO	Boolean	Indicates a fatal error – e.g., #MC during TD operation.	0x8000000000000001
TD Management	NUM_TDCX	None	RO	Unsigned Integer	Number of TDCX pages that have been added by TDH.MNG.ADDCX	0x8000000000000002

Class	Field	VMM Access		Type	Description	Field Code
		Prod.	Debug			
TD Management	TDCX_PA	None	RO	Array of Physical Address	Physical addresses of the TDCX pages (without HKID bits)	0x8000000000000010
TD Management	CHLDCNT	None	RO	64b Unsigned Integer	The number of 4KB child pages (including opaque control structure pages) associated with this TDR	0x8000000000000004
TD Management	LIFECYCLE_STATE	None	RO	LIFECYCLE_STATE	The life cycle state of this TD	0x8000000000000005
Key Management	HKID	None	RO	16b Unsigned Integer	Private HKID	0x8100000000000001
Key Management	PKG_CONFIG_ BITMAP	None	RO	Bitmap	Bitmap that indicates on which package TDH.MNG.KEY.CONFIG was executed successfully using this private key entry	0x8100000000000002

21.1.3. TDCS

Note: This section describes TDCS, as defined. Implementation may differ.

TDCS complements TDR as the logical control structure of a guest TD. TDCS is encrypted with the guest TS's ephemeral private key. It controls the guest TD operation and holds the state that is global to all the TD's VCPUs.

TDCS fields are divided into the following classes:

Table 21.4: TDCS Field Classes Definition

Field Class	Description
TD Management	These fields are used to manage the TDCS, its descendent TD private memory pages and control structure pages.
TD Execution Control	Control the execution of the guest TD: some TD execution control fields are provided as an input to TDH.MNG.INIT, and some of those are included in the TDG.MR.REPORT.
TLB Epoch Tracking	Track the TLB epoch of the guest TD – see 8.7 for details
Measurement	TD measurement registers and associated fields – see Chapter 11 for details
MSR Bitmaps	MSR bitmaps that control VM exit from the guest TD on RDMSR/WRMSR are common to all TD VCPUs and thus are stored as part of TDCS.
Secure EPT Root Page	The root page (PML5 or PML4) of the secure EPT

Note: The table below lists only TDCS fields that may be accessed by the host VMM in either production or debug mode.

Table 21.5: TDCS Definition

Class	Field	VMM Access		Guest Access	Type	Description	Base Field ID
		Prod.	Debug				
TD Management	FINALIZED	RO	RO	None	Boolean	Flags that TD build and measurement has been finalized	0x9000000000000000

Class	Field	VMM Access		Guest Access	Type	Description	Base Field ID
		Prod.	Debug				
TD Management	NUM_VCPUS	RO	RO	RO	32b Unsigned Integer	The number of VCPUs that are either in TDX non-root mode (TDVPS.STATE == VCPU_ACTIVE) or are ready to run (TDVPS.STATE == VCPU_READY): this includes VCPUs that have been successfully initialized (by TDINITVP) and have not since started teardown (due to a Triple Fault)	0x9000000000000001
TD Management	NUM_ASSOC_VCPUS	RO	RO	None	32b Unsigned Integer	The number of VCPUS associated with LPs – i.e., the LPs might hold TLB translations and/or cached TD VMCS	0x9000000000000002
Execution Controls	ATTRIBUTES	RO	RO	RO	ATTRIBUTES	TD attributes	0x1100000000000000
Execution Controls	XFAM	RO	RO	RO	XCR0	Extended Features Available Mask: indicates the extended user and system features which are available for the TD. Copied to each TDVPS on TDH.VP.INIT.	0x1100000000000001
Execution Controls	MAX_VCPUS	RO	RO	RO	32b Unsigned Integer	Maximum number of VCPUs	0x1100000000000002
Execution Controls	GPAW	RO	RO	RO	Boolean	This bit has the same meaning as the VMCS GPAW execution control: 0: GPA.SHARED bit is GPA[47] 1: GPA.SHARED bit is GPA[51]	0x1100000000000003
Execution Controls	EPTP	RO	RO	None	EPTP	TD-scope Secure EPT pointer: format is the same as the VMCS EPTP execution control; copied to each TD VMCS EPTP on TDH.VP.INIT	0x1100000000000004
Execution Controls	TSC_OFFSET	RO	RO	None	64b unsigned Integer	TD-scope TSC offset execution control: copied to each TD VMCS TSC-offset execution control on TDINITVP	0x110000000000000A
Execution Controls	TSC_MULTIPLIER	RO	RO	None	64b Unsigned Integer	TD-scope TSC multiplier execution control: copied to each TD VMCS TSC-multiplier execution control on TDH.VP.INIT	0x110000000000000B
Execution Controls	TSC_FREQUENCY	RO	RO	RO	16b Unsigned Integer	Virtual TSC frequency – in units of 25MHz	0x110000000000000C
Execution Controls	NOTIFY_ENABLES	None	RW	RW	Bitmap	Enable guest notification of events: bit 0: Notify when Zero Step attack is suspected bits 63:1: Reserved, must be 0	0x9100000000000010
Execution Controls	CPUID_VALUES	RO	RO	None	CPUID_RET	Values returned by CPUID leaves/sub-leaves	0x9100000000000400

Class	Field	VMM Access		Guest Access	Type	Description	Base Field ID
		Prod.	Debug				
Execution Controls	XBUFF_OFFSETS	RO	RO	None	Unsigned Integer	XSAVE buffer components offsets – calculated by TDH.MNG.INIT based on XFAM	0x1100000000000800
TLB Epoch Tracking	TD_EPOCH	RO	RO	None	64b Integer	The TD epoch counter: incremented by the host VMM using the TDH.MEM.TRACK function	0x9200000000000000
TLB Epoch Tracking	REFCOUNT	RO	RO	None	16b Unsigned Integer	Each REFCOUNT counts the number of LPs which may have TLB entries created during a specific TD_EPOCH and are currently executing in TDX non-root mode.	0x9200000000000001
Measurement	MRTD	RO	RO	RO	SHA384_HASH	Measurement of the initial contents of the TD	0x1300000000000000
Measurement	MRCONFIGID	RO	RO	RO	SHA384_HASH	Software-defined ID for non-owner-defined configuration of the guest TD – e.g., run-time or OS configuration	0x1300000000000010
Measurement	MROWNER	RO	RO	RO	SHA384_HASH	Software-defined ID for the guest TD's owner	0x1300000000000018
Measurement	MROWNERCONFIG	RO	RO	RO	SHA384_HASH	Software-defined ID for owner-defined configuration of the guest TD – e.g., specific to the workload rather than the run-time or OS	0x1300000000000020
Measurement	RTMR	None	RO	RO	Array of SHA384_HASH	Array of NUM_RTMRS run-time extendable measurement registers	0x1300000000000040
Measurement	MRTD_CONTEXT	None	RO	None	N/A	Non-architectural context used during ongoing calculation of MRTD until TDH.MR.FINELIZE	0x9300000000000080
MSR Bitmaps	MSR_BITMAPS	None	RO	None	64b bitmap	TD-scope RDMSR/WRMSR exit control bitmaps	0x2000000000000000
Secure EPT Root	SEPT_ROOT	None	RO	None	Secure EPT Entry	Secure EPT root page (PML5 or PML4)	0x2100000000000000

21.2. TDVPS: VCPU-Scope Control Structure

Note: This section describes TDVPS, as defined. Implementation may differ.

TDVPS is described in 5.3.1.

21.2.1. Overview

Logically, in the Intel TDX module's linear address space, TDVPS is a single structure that holds the state and control information for a single TD VCPU. The state is loaded to the LP on TD Entry and saved on TD exits.

Physically, TDVPS is composed of a root page (TDVPR) and multiple extension pages (TDVPX). The pages need not be contiguous in physical memory.

TDVPS fields are divided into the following classes:

Table 21.6: TDVPS Field Classes Definition

Field Class	Description
VCPU Management	These fields are used to manage the TDVPS and the TD VCPU.
TD VMCS	The TD VCPU's architectural VMCS
VAPIC	The TD VCPU's Virtual APIC page
VE_INFO	Holds Virtualization Exception (#VE) information
Guest GPR State	TD VCPU's general-purpose register state
Guest MSR State	TD VCPU's MSR state
Guest Extended State	TD VCPU's extended state

21.2.2. How to Read the TDVPS (including TD VMCS) Tables

21.2.2.1. *VMM Access using TDH.VP.RD and TDH.VP.WR*

Describes whether this field is accessible to the host VMM, using TDH.VP.RD and TDH.VP.WR, in production mode (ATTRIBUTES.DEBUG == 0) and debug mode (ATTRIBUTES.DEBUG == 1). Possible values are shown in the table below.

Table 21.7: VMM Access Column Definition

VMM Access	Meaning
None	No host VMM access to the field
RO	Host VMM can only read the field (using TDH.VP.RD)
RW	Host VMM can read and write the field using TDH.VP.RD and TDH.VP.WR. TDH.VP.WR does not impose any limitations except for checking that the value fits in the field size.
RWS	Host VMM can read and write the field using TDH.VP.RD and TDH.VP.WR. Writing is subject to the same limitations as if the field was modified by the guest TD (for guest state fields) and/or other limitation as described per field.

21.2.2.2. *Text Highlighting*

In the TD VMCS tables, text is highlighted to emphasize how the VMCS field value is determined and whether it can be modified during a TD VCPU life cycle, as shown in the table below.

Table 21.8: Text Highlighting in the TD VMCS Tables

Highlighting	Meaning
Black Text	Value is set by the Intel TDX module (based on a **constant** value) on TD VCPU initialization (TDH.VP.INIT), and it does not change afterwards.
Purple Text	On TD VCPU initialization (TDH.VP.INIT), value is **computed** by the Intel TDX module (e.g., based on TD parameters), and it does not change afterwards.
Blue Text	Value may be modified **by the Intel TDX module** during guest TD VCPU life cycle (possibly only in debug mode).
Grey Background	Reserved field: value is determined by the IA32_VMX_* MSRs

21.2.3. TDVPS (excluding TD VMCS)

Note: The table below lists only TDVPS fields that may be accessed by the host VMM in either production or debug mode.

Table 21.9: TDVPS Definition

Class	Field	VMM Access		Type	Description	Base Field ID
		Prod.	Debug			
Management	VCPU_STATE	None	RO	VCPU_STATE	The activity state of the VCPU.	0xA000000000000000
Management	LAUNCHED	None	RO	Boolean	A Boolean flag, indicating whether the TD VCPU has been VMLAUNCH'ed on this LP since it has last been associated with this VCPU. If TRUE, VM entry should use VMRESUME. Else, VM entry should use VMLAUNCH.	0xA000000000000001
Management	VCPU_INDEX	RO	RO	32b Unsigned Integer	Sequential index of the VCPU in the parent TD. VCPU_INDEX indicates the order of VCPU initialization (by TDINITVP), starting from 0, and is made available to the TD via TDINFO. VCPU_INDEX is in the range 0 to (TDCS.MAX_VCPUS - 1), up to 0xFFFE	0xA000000000000002
Management	NUM_TDVPX	RO	RO	Unsigned Integer	Sequential index of the VCPU in the parent TD. VCPU_INDEX indicates the order of VCPU initialization (by TDINITVP), starting from 0, and is made available to the TD via TDINFO. VCPU_INDEX is in the range 0 to (TDCS.MAX_VCPUS - 1), up to 0xFFFE	0xA000000000000003
Management	TDVPS_PAGE_PA	RO	RO	Array of PA	An array of TDVPS_PAGES physical address pointers to the TDVPS physical pages • PA is without HKID bits • Page 0 is the PA of the TDVPR page • Pages 1,2... are PAs of the TDVPX pages	0xA000000000000010
Management	ASSOC_LPID	RO	RO	Integer	The unique, hardware-derived identifier of the logical processor on which this VCPU is currently associated (either by TDENTER or by other VCPU-specific SEAMCALL flow): • A value of -1 indicates that VCPU is not associated with any LP. • Initialized by TDH.VP.INIT to the LP_ID on which it ran.	0xA000000000000004
Management	ASSOC_HKID	RO	RO	Integer	The TD's ephemeral private HKID at the last time this VCPU was associated (either by TDENTER or by other VCPU-specific SEAMCALL flow) with an LP: initialized by TDH.VP.INIT to the current TD ephemeral private HKID	0xA000000000000005

Class	Field	VMM Access		Type	Description	Base Field ID
		Prod.	Debug			
Management	VCPU_EPOCH	RO	RO	Integer	The value of TDCS.TD_EPOCH at the time this VCPU entered TDX non-root mode	0xA000000000000006
Management	CPUID_SUPERVISOR_VE	RO	RO	Boolean	When set, the Intel TDX module injects #VE on guest TD execution of CPUID in CPL = 0.	0xA000000000000007
Management	CPUID_USER_VE	RO	RO	Boolean	When set, the Intel TDX module injects #VE on guest TD execution of CPUID in CPL > 0.	0xA000000000000008
Management	IS_SHARED_EPTP_VALID	RO	RO	Boolean	Indicates that Shared EPTP is valid: set on successful TDH.VP.WR to Shared EPTP	0xA000000000000009
Management	LAST_EXIT_TSC	None	RO	Unsigned 64b Integer	Initialized to the value returned rdtsc on TDH.VP.INIT	0xA00000000000000A
Management	PEND_NMI	RW	RW	Boolean	When set, the Intel TDX module injects an NMI to the guest TD at the next available opportunity (NMI window open after TDENTER). The module then clears PEND_NIM.	0x200000000000000B
Management	XFAM	RO	RW	Bitmap	Copied from TDCS on TDH.VP.INIT. On TDH.VP.WR, checked for architectural and platform compatibility	0x200000000000000C
Management	LAST_EPF_GPA_LIST_IDX	None	RO	Unsigned Integer	Number of valid entries in LAST_EPF_GPA_LIST	0xA00000000000000D
Management	POSSIBLY_EPF_STEPPING	None	RO	Unsigned Integer	Number of possibly-legal EPT Faults (EPFs) detected so far at this TD vCPU instruction	0xA00000000000000E
Management	LAST_EPF_GPA_LIST	None	RO	GPA	Array of GPAs that caused EPF so far at this TD vCPU instruction	0xA000000000000100
VAPIC	VAPIC	None	RO	Page	Virtual APIC Page	0x0100000000000000
VE_INFO	EXIT_REASON	None	RO			0x0200000000000000
VE_INFO	VALID	None	RO		0xFFFFFFFF: valid 0x00000000: not valid	0x0200000000000001
VE_INFO	EXIT_QUALIFICATION	None	RO			0x0200000000000002
VE_INFO	GLA	None	RO			0x0200000000000003
VE_INFO	GPA	None	RO			0x0200000000000004
VE_INFO	EPTP_INDEX	None	RO			0x0200000000000005
VE_INFO	INSTRUCTION_LENGTH	None	RO			0x8200000000000010
VE_INFO	INSTRUCTION_INFORMATION	None	RO			0x8200000000000011
Guest GPR State	RAX	None	RW			0x1000000000000000
Guest GPR State	RCX	None	RW		Init value is provided as an input to TDH.VP.INIT (same value as R8)	0x1000000000000001

Class	Field	VMM Access		Type	Description	Base Field ID
		Prod.	Debug			
Guest GPR State	RDX	None	RW		Init Value: • Bits [31:00]: Same as RESET value, matches CPUID.1:EAX. CPU version information includes Family, Model and Stepping • Bits [63:32]: Set to 0	0x1000000000000002
Guest GPR State	RBX	None	RW		Init Value: • Bits [05:00]: GPAW is 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 • Bits [63:06]: Reserved for future additional details, set to 0, must be ignored by vBIOS	0x1000000000000003
Guest GPR State	RBP	None	RW			0x1000000000000005
Guest GPR State	RSI	None	RW		Init Value: • Bits [31:00]: Virtual CPU index, starting from 0 and allocated sequentially on each successful TDH.VP.INIT • Bits [63:32]: Set to 0	0x1000000000000006
Guest GPR State	RDI	None	RW			0x1000000000000007
Guest GPR State	R8	None	RW		Init value is provided as an input to TDH.VP.INIT (same value as RCX)	0x1000000000000008
Guest GPR State	R9	None	RW			0x1000000000000009
Guest GPR State	R10	None	RW			0x100000000000000A
Guest GPR State	R11	None	RW			0x100000000000000B
Guest GPR State	R12	None	RW			0x100000000000000C
Guest GPR State	R13	None	RW			0x100000000000000D
Guest GPR State	R14	None	RW			0x100000000000000E
Guest GPR State	R15	None	RW			0x100000000000000F
Guest State	DR0	None	RW			0x1100000000000000
Guest State	DR1	None	RW			0x1100000000000001
Guest State	DR2	None	RW			0x1100000000000002
Guest State	DR3	None	RW			0x1100000000000003
Guest State	DR6	None	RW			0x1100000000000006
Guest State	XCR0	None	RO			0x1100000000000020
Guest State	CR2	None	RW			0x1100000000000028
Guest State	IWK.ENCKEY	None	RO		Last KeyLocker IWK loaded	0x1100000000000040
Guest State	IWK.INTKEY	None	RO			0x1100000000000044
Guest State	IWK.FLAGS	None	RO			0x1100000000000046

Class	Field	VMM Access		Type	Description	Base Field ID
		Prod.	Debug			
Guest State	VCPU_STATE_DETAILS	RO	RO		Bit 0: VMXIP, indicates that a virtual interrupt is pending delivery, i.e. VMCS.RVI[7:4] > TDVPS.VAPIC.VPPR[7:4] Bits 63:1: Reserved, set to 0	0x9100000000000100
Guest MSR State	IA32_SPEC_CTRL	None	RW			0x1300000000000048
Guest MSR State	IA32_UMWAIT_CONTROL	None	RW			0x13000000000000E1
Guest MSR State	IA32_PERFEVTSELx	None	RW			0x1300000000000186
Guest MSR State	MSR_OFFCORE_RSPx	None	RW			0x13000000000001A6
Guest MSR State	IA32_XFD	None	RO			0x13000000000001C4
Guest MSR State	IA32_XFD_ERR	None	RO			0x13000000000001C5
Guest MSR State	IA32_FIXED_CTRx	None	RW			0x1300000000000309
Guest MSR State	IA32_PERF_METRICS	None	RW			0x1300000000000329
Guest MSR State	IA32_FIXED_CTR_CTRL	None	RW			0x130000000000038D
Guest MSR State	IA32_PERF_GLOBAL_STATUS	None	RO			0x130000000000038E
Guest MSR State	IA32_PEBS_ENABLE	None	RW			0x13000000000003F1
Guest MSR State	MSR_PEBS_DATA_CFG	None	RW			0x13000000000003F2
Guest MSR State	MSR_PEBS_LD_LAT	None	RW			0x13000000000003F6
Guest MSR State	MSR_PEBS_FRONTEND	None	RW			0x13000000000003F7
Guest MSR State	IA32_A_PMCx	None	RW			0x13000000000004C1
Guest MSR State	IA32_DS_AREA	None	RW			0x1300000000000600
Guest MSR State	IA32_XSS	None	RO			0x1300000000000DA0
Guest MSR State	IA32_LBR_DEPTH	None	RW			0x13000000000014CF
Guest MSR State	IA32_STAR	None	RO			0x1300000000002081
Guest MSR State	IA32_LSTAR	None	RO			0x1300000000002082
Guest MSR State	IA32_FMASK	None	RO			0x1300000000002084
Guest MSR State	IA32_KERNEL_GS_BASE	None	RO			0x1300000000002102
Guest MSR State	IA32_TSC_AUX	None	RW			0x1300000000002103
Guest Ext. State	XBUFF	None	RW	XSAVES buffer		0x1200000000000000

21.2.4. TD VMCS

Intel SDM, 24 Virtual Machine Control Structures

Note: This section describes TD VMCS usage, as defined. Implementation may differ.

TD VMCS is a VMX format VMCS (with TDX ISA extensions) that is stored as part of TDVPS.

21.2.4.1. *TD VMCS Guest State Area*

21.2.4.1.1. TD VMCS Guest Register State Area

Intel SDM, Vol. 3, 9.1.1 Processor State after Reset
Intel SDM, Vol. 3, 24.4.1 Guest Register State

5 **Table 21.10: TD VMCS Guest Register State Area Fields**

Field	VMM Access		Init Value (after TDH.VP.INIT)
	Prod.	Debug	
Guest CR0	None	RWS	0x0021 • Bits PE (0) and NE (5) are set to 1. • All other bits are cleared to 0. The initial value is checked for compatibility with fixed-0 and fixed-1 bits according to IA32_VMX_CR0_FIXED* MSRs, except for PG (bit 31) which is allowed to be 0 since the guest TD runs as an unrestricted guest.
Guest CR3	None	RW	0
Guest CR4	None	RWS	0x2040 • Bits MCE (6) and VMXE (13) are set to 1 • All other bits are cleared to 0. The initial value is checked for compatibility with fixed-0 and fixed-1 bits according to IA32_VMX_CR4_FIXED* MSRs.
Guest DR7	None	RW	0x00000400
Guest RSP	None	RW	0
Guest RIP	None	RW	0xFFFFFFF0
Guest RFLAGS	None	RW	0x00000002
Guest ES selector	None	RW	0
Guest CS selector	None	RW	0
Guest SS selector	None	RW	0
Guest DS selector	None	RW	0
Guest FS selector	None	RW	0
Guest GS selector	None	RW	0
Guest LDTR selector	None	RW	0
Guest TR selector	None	RW	0
Guest ES base	None	RW	0
Guest CS base	None	RW	0
Guest SS base	None	RW	0
Guest DS base	None	RW	0
Guest FS base	None	RW	0
Guest GS base	None	RW	0
Guest LDTR base	None	RW	0
Guest TR base	None	RW	0
Guest GDTR base	None	RW	0
Guest IDTR base	None	RW	0
Guest ES limit	None	RW	0xFFFFFFFF
Guest CS limit	None	RW	0xFFFFFFFF
Guest SS limit	None	RW	0xFFFFFFFF
Guest DS limit	None	RW	0xFFFFFFFF
Guest FS limit	None	RW	0xFFFFFFFF
Guest GS limit	None	RW	0xFFFFFFFF
Guest LDTR limit	None	RW	0x0000FFFF
Guest TR limit	None	RW	0x0000FFFF
Guest GDTR limit	None	RW	0x0000FFFF
Guest IDTR limit	None	RW	0
Guest ES access rights	None	RW	0x0000C093 (Data, RW, Accessed, DPL=0, Present, 32b, 4KB granularity)
Guest CS access rights	None	RW	0x0000C09B (Code, RX, Accessed, DPL=0, Present, 32b)
Guest SS access rights	None	RW	0x0000C093 (Data, RW, Accessed, DPL=0, Present, 32b, 4KB granularity)
Guest DS access rights	None	RW	0x0000C093 (Data, RW, Accessed, DPL=0, Present, 32b, 4KB granularity)

Field	VMM Access		Init Value (after TDH.VP.INIT)
	Prod.	Debug	
Guest FS access rights	None	RW	0x0000C093 (Data, RW, Accessed, DPL=0, Present, 32b, 4KB granularity)
Guest GS access rights	None	RW	0x0000C093 (Data, RW, Accessed, DPL=0, Present, 32b, 4KB granularity)
Guest LDTR access rights	None	RW	0x00010082 (LDT, Present, 32b, 1B granularity, Unusable)
Guest TR access rights	None	RW	0x0000008B (32b TSS, Busy, Present, 32b, 1B granularity)
Guest SMBASE	None	None	0

21.2.4.1.2. TD VMCS Guest MSRs

See also the MSR virtualization tables in 18.1.

Table 21.11: TD VMCS Guest MSRs

Field	VMM Access		Init Value (after TDH.VP.INIT)
	Prod.	Debug	
IA32_DEBUGCTL	None	RWS	0
IA32_SYSENTER_CS	None	RW	0
IA32_SYSENTER_ESP	None	RW	0
IA32_SYSENTER_EIP	None	RW	0
IA32_PERF_GLOBAL_CTRL	None	RW	0x000000FF • EN_PMCx (bits 0 to (NUM_PMC - 1)) are set to 1. • All other bits are cleared to 0.
IA32_PAT	None	RW	0x0007040600070406
IA32_EFER	None	RW	0x901 • SCE (bit 0) is set to 1. • LME (bit 8) is set to 1. • NXE (bit 11) is set to 1. • All other bits are cleared to 0.
GUEST_IA32_S_CET	None	RW	0
GUEST_SSP	None	RW	0
GUEST_IA32_INTERRUPT_SSP_TABLE_ADDR	None	RW	0
IA32_RTIT_CTL	None	RW	0
IA32_LBR_CTL	None	RW	0
IA32_BNDCFGS	None	RO	0
IA32_GUEST_PKRS	None	RW	0

5

21.2.4.1.3. TD VMCS Guest Non-Register State Area

Intel SDM, 24.4.2 Guest Non-Register State

Table 21.12: TD VMCS Guest Non-Register State Area Fields

Field Name	VMM Access		Description	Initial State
	Prod.	Debug		
Activity State	None	RO	Saved/restored on VM exit/entry	Active (0)
Interruptibility State	None	RW	Saved/restored on VM exit/entry	0
Pending Debug Exceptions	None	RW	Saved/restored on VM exit/entry	0
VMCS Link Pointer	None	None	Saved/restored on VM exit/entry	NULL_PA (-1)
VMX-Preemption Timer Value	None	RW	N/A: VMX-preemption timer is not used by guest TDs.	0

Field Name	VMM Access		Description	Initial State
	Prod.	Debug		
PDPTEn	None	RW	N/A: PAE paging is not used by TD guests.	NULL_PA (-1)
Guest Interrupt Status	None	RW	Includes RVI (lower byte) and SVI (upper byte): saved/restored on VM exit/entry	0
PML Index	None	RW	N/A: PML is not used by guest TDs.	0
Guest UINV	None	RW		0

21.2.4.2. *TD VMCS Host State Area*

Intel SDM, 24.5 Host-State Area

The host state area is not intended to be accessible outside the Intel TDX module.

21.2.4.3. *TD VMCS VM-Execution Control Fields*

Intel SDM, 24.6 VM-Execution Control Fields

21.2.4.3.1. TD VMCS Pin-Based VM-Execution Controls

Table 21.13: TD VMCS Pin-Based VM-Execution Controls

Bit	Name	VMM Access		Description	Init Value
		Prod.	Debug		
0	External-interrupt exiting	None	RO	The Intel TDX module performs TD Exit	1
1	Reserved	None	RO		MSR
2	Reserved	None	RO		MSR
3	NMI exiting	None	RO	The Intel TDX module performs TD Exit	1
4	Reserved	None	RO		MSR
5	Virtual NMIs	None	RO		1
6	Activate VMX-preemption timer	None	RO		0
7	Process posted interrupts	RWS	RWS	Set to 1 by TDH.VP.WR only if a valid posted interrupt descriptor and a valid posted interrupt notification vector are set.	0
8	Reserved	None	RO		MSR
9	Reserved	None	RO		MSR
10	Reserved	None	RO		MSR
11	Reserved	None	RO		MSR
12	Reserved	None	RO		MSR
13	Reserved	None	RO		MSR
14	Reserved	None	RO		MSR
15	Reserved	None	RO		MSR
16	Reserved	None	RO		MSR
17	Reserved	None	RO		MSR
18	Reserved	None	RO		MSR
19	Reserved	None	RO		MSR
20	Reserved	None	RO		MSR
21	Reserved	None	RO		MSR
22	Reserved	None	RO		MSR
23	Reserved	None	RO		MSR
24	Reserved	None	RO		MSR
25	Reserved	None	RO		MSR
26	Reserved	None	RO		MSR
27	Reserved	None	RO		MSR

		VMM Access			
Bit	Name	Prod.	Debug	Description	Init Value
28	Reserved	None	RO		MSR
29	Reserved	None	RO		MSR
30	Reserved	None	RO		MSR
31	Reserved	None	RO		MSR

Reserved bits are set based on IA32_VMX_TRUE_PINBASED_CTLS MSR.

21.2.4.3.2. TD VMCS Processor-Based VM-Execution Controls

Table 21.14: TD VMCS Primary Processor-Based VM-Execution Controls

		VMM Access			
Bit	Name	Prod.	Debug	Description	Init Value
0	Reserved	None	RO		MSR
1	Reserved	None	RO		MSR
2	Interrupt-window exiting	None	RW		0
3	Use TSC offsetting	None	RO		1
4	Reserved	None	RO		MSR
5	Reserved	None	RO		MSR
6	Reserved	None	RO		MSR
7	HLT exiting	None	RO	The Intel TDX module injects a #VE into the guest TD	1
8	Reserved	None	RO		MSR
9	INVLPG exiting	None	RW		0
10	MWAIT exiting	None	RO	The Intel TDX module injects a #VE into the guest TD	1
11	RDPMC exiting	None	RW		~TDCS.ATTRIBUTES.PERFMON
12	RDTSC exiting	None	RW		0
13	Reserved	None	RO		MSR
14	Reserved	None	RO		MSR
15	CR3-load exiting	None	RW		0
16	CR3-store exiting	None	RW		0
17	Activate tertiary controls	None	RO		1
18	Reserved	None	RO		MSR
19	CR8-load exiting	None	RW		0
20	CR8-store exiting	None	RW		0
21	Use TPR shadow	None	RO		1
22	NMI-window exiting	None	RO	Set by the Intel TDX module before entering the guest TD – based on TDVPS.PEND_NMI	0
23	MOV-DR exiting	None	RW		0
24	Unconditional I/O exiting	None	RW		1
25	Use I/O bitmaps	None	RO		0
26	Reserved	None	RO		MSR
27	Monitor trap flag	None	RW		0
28	Use MSR bitmaps	None	RO		1
29	MONITOR exiting	None	RW		1
30	PAUSE exiting	None	RW		0
31	Activate secondary controls	None	RO		1

5

Reserved bits are set based on IA32_VMX_TRUE_PROCBASED_CTLS MSR.

Table 21.15: TD VMCS Secondary Processor-Based VM-Execution Controls

		VMM Access			
Bit	Name	Prod.	Debug	Description	Init Value
0	Virtualize APIC accesses	None	RO		0

		VMM Access			
Bit	Name	Prod.	Debug	Description	Init Value
1	Enable EPT	None	RO		1
2	Descriptor-table exiting	None	RW		0
3	Enable RDTSCP	None	RO		1
4	Virtualize x2APIC mode	None	RO		1
5	Enable VPID	None	RO		1
6	WBINVD exiting	None	RO		1
7	Unrestricted guest	None	RO		1
8	APIC-register virtualization	None	RO		1
9	Virtual-interrupt delivery	None	RO		1
10	PAUSE-loop exiting	None	RW		0
11	RDRAND exiting	None	RW		0
12	Enable INVPCID	None	RO		1
13	Enable VM functions	None	RO		1
14	VMCS shadowing	None	RO		0
15	Enable ENCLS exiting	None	RO		1
16	RDSEED exiting	None	RW		0
17	Enable PML	None	RWS	If set to 1, PML address must be a valid shared physical address	0
18	EPT-violation #VE	None	RO		1
19	Conceal VMX from PT	None	RO		1
20	Enable XSAVES/XRSTORS	None	RW		1
21	PASID translation	None	RO		1
22	Mode-based execute control for EPT	None	RO		0
23	Enable SPP	None	RO		0
24	PT uses guest physical addresses (PT2GPA)	None	RO		1
25	Use TSC scaling	None	RO		1
26	Enable user-level wait and pause	None	RO		Set to the value of virtualized CPUID(0x7,0x0).ECX[5]
27	Enable PCONFIG	None	RO		Set to the value of virtualized CPUID(0x7,0x0).EDX[18]
28	Enable ENCLV exiting	None	RO		1
29	Enable EPC Virtualization Extensions	None	RO		0
30	Bus-lock detection	RW	RW	If enabled by the host VMM (using TDH.VP.WR), then the Intel TDX module performs TD Exit on VM exit.	0
31	Notification exiting	RW	RW	If enabled by the host VMM (using TDH.VP.WR), then the Intel TDX module performs TD Exit on VM exit.	0

Table 21.16: TD VMCS Tertiary Processor-Based VM-Execution Controls

		VMM Access			
Bit	Name	Prod.	Debug	Description	Init Value (after TDH.VP.INIT)
0	LOADIWKEY exiting	None	RW		0
1	Enable HLAT	None	RO		0
2	EPT paging-write control	None	RO		0
3	Guest paging verification	None	RO		0
4	IPI virtualization	None	RO		0
5	GPAW	None	RO	0: GPA.SHARED bit is GPA[47] 1: GPA.SHARED bit is GPA[51]	Copied from TDCS.GPAW

		VMM Access			
Bit	**Name**	**Prod.**	**Debug**	**Description**	**Init Value (after TDH.VP.INIT)**
6	Reserved	None	RO		MSR
7	Reserved	None	RO		MSR
8	Reserved	None	RO		MSR
9	Reserved	None	RO		MSR
10	Reserved	None	RO		MSR
11	Reserved	None	RO		MSR
12	Reserved	None	RO		MSR
13	Reserved	None	RO		MSR
14	Reserved	None	RO		MSR
15	Reserved	None	RO		MSR
16	Reserved	None	RO		MSR
17	Reserved	None	RO		MSR
18	Reserved	None	RO		MSR
19	Reserved	None	RO		MSR
20	Reserved	None	RO		MSR
21	Reserved	None	RO		MSR
22	Reserved	None	RO		MSR
23	Reserved	None	RO		MSR
24	Reserved	None	RO		MSR
25	Reserved	None	RO		MSR
26	Reserved	None	RO		MSR
27	Reserved	None	RO		MSR
28	Reserved	None	RO		MSR
29	Reserved	None	RO		MSR
30	Reserved	None	RO		MSR
31	Reserved	None	RO		MSR
32	Reserved	None	RO		MSR
33	Reserved	None	RO		MSR
34	Reserved	None	RO		MSR
35	Reserved	None	RO		MSR
36	Reserved	None	RO		MSR
37	Reserved	None	RO		MSR
38	Reserved	None	RO		MSR
39	Reserved	None	RO		MSR
40	Reserved	None	RO		MSR
41	Reserved	None	RO		MSR
42	Reserved	None	RO		MSR
43	Reserved	None	RO		MSR
44	Reserved	None	RO		MSR
45	Reserved	None	RO		MSR
46	Reserved	None	RO		MSR
47	Reserved	None	RO		MSR
48	Reserved	None	RO		MSR
49	Reserved	None	RO		MSR
50	Reserved	None	RO		MSR
51	Reserved	None	RO		MSR
52	Reserved	None	RO		MSR
53	Reserved	None	RO		MSR
54	Reserved	None	RO		MSR
55	Reserved	None	RO		MSR
56	Reserved	None	RO		MSR
57	Reserved	None	RO		MSR
58	Reserved	None	RO		MSR
59	Reserved	None	RO		MSR

Bit	Name	VMM Access		Description	Init Value (after TDH.VP.INIT)
		Prod.	Debug		
60	Reserved	None	RO		MSR
61	Reserved	None	RO		MSR
62	Reserved	None	RO		MSR
63	Reserved	None	RO		MSR

Reserved bits are set based on IA32_VMX_PROCBASED_CTLS3 MSR.

21.2.4.3.3. TD VMCS Controls for APIC Virtualization

Table 21.17: TD VMCS Controls for APIC Virtualization

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
APIC-access address	None	RO		NULL_PA (-1)
Virtual-APIC address	None	None	On VCPU-to-LP association, set by the Intel TDX module to the address of the VAPIC page in TDVPS, including the TD's ephemeral HKID	Address of the VAPIC page in TDVPS, including the TD's ephemeral HKID
TPR threshold	None	RO		0
EOI-exit bitmap n	None	RO		0
Posted-interrupt notification vector	RWS	RWS	TDH.VP.WR checks the value to be in the range 0 to 255. See process posted interrupt pin-based execution control.	0xFFFF
Posted-interrupt descriptor address	RWS	RWS	TDH.VP.WR checks the value as follows: • It must be a valid shared physical address (HKID bits encode a shared HKID). • It must be aligned on 64B. See process posted interrupt pin-based execution control.	0xFFFFFFFFFFFFFFC0

5

21.2.4.3.4. EPTP and Shared EPTP

Table 21.18: EPTP (Copied from TDCS.EPTP on TDH.VP.INIT)

Bits	Field Name	VMM Access		Description	Initial Value
		Prod.	Debug		
2:0	EPT Memory Type	RO	RO	Set to WB	6
5:3	EPT Level	RO	RO	1 less than the EPT page-walk length	Copied from TDCS.EPTP
6	Enable A/D Bits	RO	RO		0
7	Enable supervisor shadow stack control	RO	RO		0
11:8	Reserved	RO	RO		0
51:12	EPML5/4 PA	RO	RO		
63:52	Reserved	RO	RO		0

Table 21.19: Shared EPTP

Bits	Field Name	VMM Access		Description	Initial Value
		Prod.	Debug		
11:0	Reserved	None	RO		0
51:12	EPML5/4 PA	RWS	RWS		
63:52	Reserved	None	RO		0

21.2.4.3.5. CR-Related TD VMCS VM-Execution Control Fields

Table 21.20: CR-Related VMCS VM-Execution Control Fields

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
CR0 Guest/Host Mask	None	RW	Bits 0, 5, 29 and 30 can't be written even in debug mode	The following bits are set to 1, indicating they are owned by the Intel TDX module: • PE (0) • NE (5) • NW (29) • CD (30) • Any bit set to 1 in IA32_VMX_CR0_FIXED0 (i.e., a bit whose value must be 1) • Any bit set to 0 in IA32_VMX_CR0_FIXED1 (i.e., a bit whose value must be 0), except for PG(31) which is set to 0, since the guest TD runs as an unrestricted guest • Bits known to the Intel TDX module as reserved (bits 63-32, 28-19, 17 and 15-6) All other bits are cleared to 0, indicating they are owned by the guest TD.
CR0 Read Shadow	None	RW	Bits 0 and 5 can't be written even in debug mode	The following bits are set to 1: • PE (0) • NE (5) • Any bit set to 1 in IA32_VMX_CR0_FIXED0 (i.e., a bit whose value must be 1) All other bits are cleared to 0.

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
CR4 Guest/Host Mask	None	RW	Bits 6, 13 and 14 can't be written even in debug mode	• Bits MCE (6), VMXE (13) and SMXE (14) are set to 1, indicating they are owned by the Intel TDX module. • Bit PKE (22) is set to ~TDCS.XFAM[9] to intercept writes to CR4 If PK is not enabled. • If TDCS.XFAM[12:11] is 11, then bit CET (23) is cleared to 0. Otherwise (CET is not enabled), bit CET (23) is set to 1 to intercept writes to CR4. • Bit UINT (25) is set ~TDCS.XFAM[14] to intercept writes to CR4 if ULI is not enabled. • Bit KL (19) is set to ~TDCS.ATTRIBUTES.KL to intercept writes to CR4 if KeyLocker is not enabled. • Bit PKS (24) is set to ~TDCS.ATTRIBUTES.PKS to intercept writes to CR4 if PKS is not enabled. • Any bit set to 1 in IA32_VMX_CR4_FIXED0 (i.e., a bit whose value must be 1) is set to 1. • Any bit set to 0 in IA32_VMX_CR4_FIXED1 (i.e., a bit whose value must be 0) is set to 1. • Bits known to the Intel TDX module as reserved (bits 63-26 and bit 15) are set to 1. • All other bits are cleared to 0.
CR4 Read Shadow	None	RW	Bit 6 can't be written even in debug mode	• Bit MCE (6) is set to 1. • Bit VMXE (13) is cleared to 0. • Any other bit whose value is set to 1 in IA32_VMX_CR4_FIXED0 (i.e., a bit whose value must be 1) is set to 1. • All other bits are cleared to 0.
CR3-Target Values	None	RW	N/A: The Intel TDX module does not control guest CR3	N/A
CR3-Target Count	None	RW	Set to 0: Intel TDX module does not control guest CR3	0

21.2.4.3.6. Other TD VMCS VM-Execution Control Fields

Table 21.21: Other TD VMCS VM-Execution Control Fields

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
Exception Bitmap	None	RW	• Bit 18 (MCE) is set to 1, even in debug mode. • Other bits are cleared to 0. They may be modified in debug mode.	0x00040000
Page-fault error-code mask	None	RW		0
Page-fault error-code match	None	RW		0
I/O-Bitmap Address n	None	RO	Set to NULL_PA (-1): I/O bitmaps execution control is set to 0	NULL_PA (-1)

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
Time-Stamp Counter Offset	RO	RW		Copied from TDCS.TSC_OFFSET
Time-Stamp Counter Multiplier	RO	RW		Copied from TDCS.TSC_MULTIPLIER
MSR-Bitmap Address	RO	RO		
Executive-VMCS Pointer	None	None	N/A	NULL_PA (-1)
TD HKID	RO	RO		
VPID	None	RO	1 + the sequential initialization index of the VCPU (TDVPS.VCPU_INDEX + 1)	Set to 1 + the sequential index of the VCPU (TDVPS.VCPU_INDEX + 1)
PLE_GAP	RO	RW		0
PLE_Window	RO	RW		0
VM-Function Controls	RO	RO	The Intel TDX module injects a #UD into the TD.	0
EPTP-list address	RO	RO	VMFUNC is not supported.	NULL_PA (-1)
VMREAD-bitmap address	None	RO	VMCS shadowing is not supported.	NULL_PA (-1)
VMWRITE-bitmap address	None	RO	VMCS shadowing is not supported.	NULL_PA (-1)
ENCLS-Exiting Bitmap	None	RO	Set to all 1's – the Intel TDX module injects a #UD into the guest TD.	All 1s
ENCLV-Exiting Bitmap	None	RO	Set to all 1's – the Intel TDX module injects a #UD into the guest TD.	All 1s
PML address	RO	RWS	TDH.VP.WR checks the value as follows: • It must be a valid shared physical address (HKID bits encode a shared HKID). • It must be aligned on 4KB. See enable PML execution control.	0xFFFFFFFFFFFFF000
Virtualization-exception information address	None	RO		
EPTP index	None	RO		0
XSS-Exiting Bitmap	None	RW		0
low PASID directory address	None	RO		Implementation-dependent
high PASID directory address	None	RO		Implementation-dependent
notify window	RW	RW		0
PCONFIG-Exiting Bitmap	None	RO		-1

21.2.4.4. *TD VMCS VM-Exit Control Fields*

Intel SDM, 24.7 VM-Exit Control Fields

Table 21.22: TD VMCS VM-Exit Controls

	VMM Access			
Name	**Prod.**	**Debug**	**Description**	**Init Value**
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Save debug controls	None	RO		1
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Host address-space size	None	RO		1
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Load IA32_PERF_GLOBAL_CTRL	None	RO		Set to 1 if TDCS.ATTRIBUTES.PERFMON = 1 or ATTRIBUTES.DEBUG = 1
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Acknowledge interrupt on exit	None	RO		1
Reserved	None	RO		MSR
Reserved	None	RO		MSR
Save IA32_PAT	None	RO		1
Load IA32_PAT	None	RO		1
Save IA32_EFER	None	RO		1
Load IA32_EFER	None	RO		1
Save VMX-preemption time value	None	RO		Set to 1 if TDCS.ATTRIBUTES.DEBUG = 1
Clear IA32_BNDCFGS	None	RO	Deprecated	0
Conceal VMX from PT	None	RO		1
Clear IA32_RTIT_CTL	None	RO		1
Clear IA32_LBR_CTL	None	RO		1
Clear UINV	None	RO		1
Load host CET state	None	RO		1
Load host PKRS	None	RO		0
Save IA32_PERF_GLOBAL_CTRL	None	RO		Set to 1 if TDCS.ATTRIBUTES.PERFMON = 1 or ATTRIBUTES.DEBUG = 1
Reserved	None	RO		MSR

5

Reserved bits are set based on IA32_VMX_TRUE_EXIT_CTLS MSR.

Table 21.23: TD VMCS VM-Exit Controls for MSRs

Field Name	**VMM Access**		**Description**	**Initial Value**
	Prod.	**Debug**		
VM-exit MSR-store count	None	RO	Not used	0
VM-exit MSR-store address	None	RO	Not used	NULL_PA (-1)
VM-exit MSR-load count	None	RW	Not used	0
VM-exit MSR-load address	None	RO	Not used	NULL_PA (-1)

21.2.4.5. *TD VMCS VM-Entry Control Fields*

Intel SDM, 24.8 VM-Entry Control Fields

Table 21.24: TD VMCS VM-Entry Controls

		VMM Access			
Bit	**Name**	**Prod.**	**Debug**	**Description**	**Init Value (after TDH.VP.INIT)**
0	Reserved	None	RO		MSR
1	Reserved	None	RO		MSR
2	Load debug controls	None	RO		1
3	Reserved	None	RO		MSR
4	Reserved	None	RO		MSR
5	Reserved	None	RO		MSR
6	Reserved	None	RO		MSR
7	Reserved	None	RO		MSR
8	Reserved	None	RO		MSR
9	IA-32e mode guest	None	RO	Written by the CPU on VM exit	0
10	Entry to SMM	None	RO		0
11	Deactivate dual-monitor treatment	None	RO		0
12	Reserved	None	RO		MSR
13	Load IA32_PERF_GLOBAL_CTRL	None	RO		Set to 1 if TDCS.ATTRIBUTES.PERFMON = 1 or ATTRIBUTES.DEBUG = 1
14	Load IA32_PAT	None	RO		1
15	Load IA32_EFER	None	RO		1
16	Load IA32_BNDCFGS	None	RO		0
17	Conceal VMX from PT	None	RO		1
18	Load IA32_RTIT_CTL	None	RO		1
19	Load UINV	None	RO		1
20	Load CET state	None	RO		1
21	Load IA32_LBR_CTL	None	RO		1
22	Load guest PKRS	None	RO		Set to 1 if TDCS.ATTRIBUTES.PKRS = 1 or TDCS.ATTRIBUTES.DEBUG = 1
23	Reserved	None	RO		MSR
24	Reserved	None	RO		MSR
25	Reserved	None	RO		MSR
26	Reserved	None	RO		MSR
27	Reserved	None	RO		MSR
28	Reserved	None	RO		MSR
29	Reserved	None	RO		MSR
30	Reserved	None	RO		MSR
31	Reserved	None	RO		MSR

5 Reserved bits are set based on IA32_VMX_ENTRY_CTLS MSR.

Table 21.25: TD VMCS VM-Entry Controls for MSRs

Field Name	**VMM Access**		**Description**	**Initial Value**
	Prod.	**Debug**		
VM-entry MSR-load count	None	RO	Not used	0
VM-entry MSR-load address	None	RO	Not used	NULL_PA (-1)

Table 21.26: TD VMCS VM-Entry Controls for Event Injection

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
VM-entry interruption information	None	RO		
VM-entry exception error code	None	RO		
VM-entry instruction length	None	RO		

21.2.4.6. *TD VMCS VM-Exit Information Fields*

Intel SDM, 24.9 VM-Exit Information Fields

Table 21.27: TD VMCS Basic VM-Exit Information

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
Exit reason	None	RO	If the Intel TDX module decides to perform a TD exit, it returns this in RAX bits 31:0. Bit 27 (enclave mode) is not set. Bit 28 (Pending MTF VM exit) is not set. Bit 29 (VM exit from VMX root operation) is not set. Bit 31 (VM-entry failure) is not set.	N/A
Exit qualification	None	RO	If the Intel TDX module decides to perform a TD exit, it returns this in RCX. If the exit is due to EPT violation, bits 12-7 of the exit qualification are cleared to 0.	N/A
Guest-Linear Address	None	RO		N/A
Guest-physical Address	None	RO	If the Intel TDX module decides to perform a TD exit, it returns this in R8. It the EPT fault was caused by an access attempt to a private page, the Intel TDX module clears bits 11:0 to 0.	N/A

Table 21.28: TD VMCS Information for VM Exits Due to Vectored Events

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
VM-exit interruption information	None	RO	On asynchronous TD exit, the Intel TDX module returns this in R9. Bits 63:32 are cleared to 0.	N/A
VM-exit interruption error code	None	RO		N/A

Table 21.29: TD VMCS Information for VM Exits That Occur During Event Delivery

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
IDT-vectoring information	None	RO		
IDT-vectoring error code	None	RO		

Table 21.30: TD VMCS Information for VM Exits Due to Instruction Execution

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
VM-exit instruction length	None	RO		
VM-exit instruction information	None	RO		
I/O RCX	None	RO		N/A
I/O RSI	None	RO		N/A
I/O RDI	None	RO		N/A
I/O RIP	None	RO		N/A

Table 21.31: TD VMCS VM-Instruction Error Field

Field Name	VMM Access		Description	Initial Value
	Prod.	Debug		
VM-instruction error	None	RO		N/A

22.ABI Reference: Interface Functions

22.1. How to Read the Interface Function Definitions

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

A table of operands is provided for any function that has explicit and/or implicit memory operands or implicit resources. Table 22.1 below describes how to read it. Most of the background is detailed in Chapter 17.

Table 22.1: How to Read the Operands Information Tables

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Alignment Check	Concurrency Restrictions		
								Resource	Contain. 2MB	Contain. 1GB
The operand may be specified explicitly or may be implicit, see 17.2 0	Register used as a pointer to the operand	HPA or GPA, see 17.2	Resource (memory or CPU internal) for this operand	Data type of the resource, as defined in Chapter 20 or Chapter 21	Type of memory or resource access: R, RW, or Ref,0 see 17.2	Shared, Private, Opaque or Hidden, see 17.2	Required alignment of the operand	Concurrency restrictions are described in 17.1. For explicit memory accesses using HPA, there are additional concurrency restrictions on the 1GB and 2MB blocks that contain the accessed HPA. For other types of accesses, only the operand concurrency is applicable. Shared(i) and Exclusive(i) indicate that the resource is implicitly restricted.		

22.2. Host-Side (SEAMCALL) Interface Functions

The SEAMCALL instruction enters the Intel TDX module. It is designed to call host-side Intel TDX functions, either local or a TD entry to a guest TD, as selected by RAX.

22.2.1. SEAMCALL Instruction (Common)

This section describes the common functionality of SEAMCALL. Leaf functions are described in the following sections.

Table 22.2: SEAMCALL Input Operands Definition

Parameter	Description
RAX	Leaf number: see Table 22.4 below.
Other	See individual SEAMCALL leaf functions.

Table 22.3: SEAMCALL Output Operands Definition

Parameter	Description
RAX	Instruction return code, indicating the outcome of execution of the instruction. See 17.3.2 for details.
Other	See individual SEAMCALL leaf functions.

Table 22.4: SEAMCALL Instruction Leaf Numbers Definition

Leaf Number	Interface Function Name
0	TDH.VP.ENTER
1	TDH.MNG.ADDCX
2	TDH.MEM.PAGE.ADD
3	TDH.MEM.SEPT.ADD
4	TDH.VP.ADDCX
5	
6	TDH.MEM.PAGE.AUG
7	TDH.MEM.RANGE.BLOCK
8	TDH.MNG.KEY.CONFIG
9	TDH.MNG.CREATE
10	TDH.VP.CREATE
11	TDH.MNG.RD
12	TDH.MEM.RD
13	TDH.MNG.WR
14	TDH.MEM.WR
15	TDH.MEM.PAGE.DEMOTE
16	TDH.MR.EXTEND
17	TDH.MR.FINALIZE
18	TDH.VP.FLUSH
19	TDH.MNG.VPFLUSHDONE
20	TDH.MNG.KEY.FREEID
21	TDH.MNG.INIT
22	TDH.VP.INIT
23	TDH.MEM.PAGE.PROMOTE
24	TDH.PHYMEM.PAGE.RDMD
25	TDH.MEM.SEPT.RD
26	TDH.VP.RD
27	TDH.MNG.KEY.RECLAIMID
28	TDH.PHYMEM.PAGE.RECLAIM
29	TDH.MEM.PAGE.REMOVE
30	TDH.MEM.SEPT.REMOVE
31	TDH.SYS.KEY.CONFIG
32	TDH.SYS.INFO
33	TDH.SYS.INIT
35	TDH.SYS.LP.INIT
36	TDH.SYS.TDMR.INIT
38	TDH.MEM.TRACK

Leaf Number	Interface Function Name
39	TDH.MEM.RANGE.UNBLOCK
40	TDH.PHYMEM.CACHE.WB
41	TDH.PHYMEM.PAGE.WBINVD
42	Reserved
43	TDH.VP.WR
44	TDH.SYS.LP.SHUTDOWN
45	TDH.SYS.CONFIG

Instruction Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

On entry, the Intel TDX module performs the checks listed below at a high level. Errors cause a SEAMRET with RAX set to the proper completion status code.

1. The leaf number in RAX is supported by the Intel TDX module.
2. If the Intel TDX module's state is not SYS_READY (see 13.1.2), only TDH.SYS.INFO, TDH.SYS.INIT, TDH.SYS.LP.INIT, TDH.SYS.CONFIG, TDH.SYS.KEY.CONFIG and TDH.SYS.SHUTDOWN leaf functions are allowed. Those leaf functions then perform other initialization state checks.

If all checks pass, the Intel TDX module calls the leaf function according to the leaf number in RAX. See the following sections for individual leaf function details.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.5: SEAMCALL Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_SUCCESS	SEAMCALL is successful.
TDX_SYS_SHUTDOWN	
Other	See individual leaf functions.

22.2.2. TDH.MEM.PAGE.ADD Leaf

Add a 4KB private page to a TD, mapped to the specified GPA, filled with the given page image and encrypted using the TD ephemeral key, and update the TD measurement with the page properties.

Table 22.6: TDH.MEM.PAGE.ADD Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the EPT entry that will map the new page – see 20.5.1: must be 0
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address to be mapped for the new Secure EPT page
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		
R8	Host physical address of the target page to be added to the TD (HKID bits must be 0)		
R9	Host physical address (including HKID bits) of the source page image		

Table 22.7: TDH.MEM.PAGE.ADD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.PAGE.ADD adds a 4KB private page to a TD and maps it to the provided GPA. It copies the provided source page image to specified physical page using the TD's ephemeral private key and updates the TD measurement with the page properties. TDH.MEM.PAGE.ADD is used during TD build before the TD is initialized.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.8: TDH.MEM.PAGE.ADD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA	TD private page (GPA)[8]	Blob	RW	Private	4KB	N/A	N/A	N/A
Explicit	RDX	HPA	TDR page	Blob	RW	Opaque	4KB	Exclusive	Shared	Shared
Explicit	R8	HPA	TD private page (HPA)[8]	Blob	RW	Private	4KB	Exclusive	Shared	Shared
Explicit	R9	HPA	Source page	Blob	R	Shared	4KB	None	None	None
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Exclusive(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.PAGE.ADD checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD build and measurement must not have been finalized (by TDH.MR.FINALIZE).
6. The target page metadata in PAMT must be correct (PT must be PT_NDA).

If successful, the function does the following:

7. Walk the Secure EPT based on the GPA operand, and find the leaf EPT entry for the 4KB page.

If the Secure EPT entry is marked as SEPT_FREE, the function does the following:

8. Copy the source image to the target TD page using the TD's ephemeral private HKID, and direct write (MOVDIR64B).
9. Update the parent Secure EPT entry with the target page HPA and SEPT_PRESENT state.
10. Extend TDCS.MRTD with the target page GPA. Extension is done using SHA384 with a 128B extension buffer composed as follows:
    - Bytes 0 through 11 contain the ASCII string "MEM.PAGE.ADD".
    - Bytes 16 through 23 contain the GPA (in little-endian format).
    - All the other bytes contain 0.
11. Increment TDR.CHLDCNT.
12. Update the PAMT entry with the PT_REG page type and the TDR physical address as the OWNER.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.9: TDH.MEM.PAGE.ADD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_FREE	

[8] RCX and R8 denote the same TD private page operand, using HPA and GPA respectively

Completion Status Code	Description
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.PAGE.ADD is successful
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_FINALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.3. TDH.MEM.PAGE.AUG Leaf

Dynamically add a 4KB or a 2MB private page to an initialized TD, mapped to the specified GPAs.

Table 22.10: TDH.MEM.PAGE.AUG Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the EPT entry that will map the new page – see 20.5.1: must be 0 (4KB) or 1 (2MB)
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address to be mapped for the new Secure EPT page
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		
R8	Host physical address of the target page to be added to the TD (HKID bits must be 0)		

Table 22.11: TDH.MEM.PAGE.AUG Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.PAGE.AUG adds a 4KB or a 2MB private page to a TD and maps it to the provided GPA. The new page is mapped in a pending state and can be accessed only by the guest TD after it accepts it using TDCALL(TDG.MEM.PAGE.ACCEPT). TDH.MEM.PAGE.AUG does not initialize the new page and does not update the TD measurement.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.12: TDH.MEM.PAGE.AUG Memory Operands Information Definition

Explicit/ Implicit	Register	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA	TD private page (GPA)[9]	Blob	None	Private	$2^{12+9*Level}$ Bytes	N/A	N/A	N/A
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Explicit	R8	HPA	TD private page (HPA)[9]	Blob	None	Private	$2^{12+9*Level}$ Bytes	Exclusive	Shared[10]	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.PAGE.AUG checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The target page metadata in PAMT must be correct (PT must be PT_NDA for the entire 4KB or 2MB range).

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and find the leaf EPT entry for the 4KB or 2MB page.

If the Secure EPT entry is marked as SEPT_FREE, the function does the following:

7. Update the parent Secure EPT entry with the target page HPA and SEPT_PENDING state.
8. Atomically increment TDR.CHLDCNT by 1 (for a 4KB page) or by 512 (for a 2MB page).
9. Update the PAMT entry with the PT_REG page type and the TDR physical address as the OWNER.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.13: TDH.MEM.PAGE.AUG Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_FREE	
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	

[9] RCX and R8 denote the same TD private page operand, using HPA and GPA respectively

[10] Applicable for 4KB pages only

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.PAGE.AUG is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_FINALIZED	
TDX_TD_NOT_INITIALIZED	

22.2.4. TDH.MEM.PAGE.DEMOTE Leaf

Split a large private TD page (2MB or 1GB) into 512 small pages (4KB or 2MB, respectively).

Table 22.14: TDH.MEM.PAGE.DEMOTE Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that maps the large page to be split: either 1 (2MB) or 2 (1GB) – see 20.5.1
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address of the large page to be split Depending on the level, the following least significant bits must be 0: Level 1 (2MB): Bits 20:12 Level 2 (1GB): Bits 29:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		
R8	Host physical address of the new Secure EPT page to be added to the TD (HKID bits must be 0)		

Table 22.15: TDH.MEM.PAGE.DEMOTE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.PAGE.DEMOTE splits a large TD private page (2MB or 1GB) into 512 small pages (4KB or 2MB, respectively) and adds a new Secure EPT page to map those small pages.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.16: TDH.MEM.PAGE.DEMOTE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource Name	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	TD private page to split	Blob	None	Private	$2^{12+9*level}$ bytes	Exclusive	None	None
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Explicit	R8	HPA	New Secure EPT page	SEPT_PAGE	RW	Private	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT Tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.PAGE.DEMOTE checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified page level is either 1 (2MB) or 2 (1GB). See 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand and locate the large TD private page to be demoted.
7. Check the page is blocked (its parent Secure EPT entry is a leaf entry, and its state is either SEPT_BLOCKED or SEPT_PENDING_BLOCKED).
8. Check that TLB tracking has been done, based on the large TD private page's PAMT.BEPOCH.

If successful, the function does the following:

9. Split the large TD private page PAMT entry into 512 PAMT entries at the lower level:
   9.1. Set the parent PAMT_2M or PAMT_1G entry state to PT_NDA.
   9.2. Set the 512 child PAMT4K or PAMT_2M entries respectively to PT_REG.
10. Initialize the new Secure EPT page's 512 entries to SEPT_PRESENT state pointing to the 512 consecutive small pages above. Use the TD's ephemeral private HKID and direct write (MOVDIR64B).
11. Atomically set the demoted Secure EPT entry to SEPT_PRESENT (if it was SEPT_BLOCKED) or SEPT_PENDING (if it was SEPT_PENDING_BLOCKED) non-leaf entry pointing to the new Secure EPT page.
12. Atomically increment TDR.CHLDCNT by 1.
    12.1. Note that CHLDCNT counts the number of 4KB pages. The change is due only to the addition of the new Secure EPT page.
13. Update the PAMT entry of the new Secure-EPT page with the PT_EPT page type and the TDR physical address as the OWNER.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.17: TDH.MEM.PAGE.DEMOTE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_LEAF	
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_NOT_BLOCKED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.PAGE.DEMOTE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TLB_TRACKING_NOT_DONE	

22.2.5. TDH.MEM.PAGE.PROMOTE Leaf

Merge 512 consecutive small private TD pages (4KB or 2MB) into one large page (2MB or 1GB, respectively).

Table 22.18: TDH.MEM.PAGE.PROMOTE Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that will map the merged large page: either 1 (2MB) or 2 (1GB) (see 20.5.1)
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address of the merged large page Depending on the level, the following least significant bits must be 0: Level 1 (2MB): Bits 20:12 Level 2 (1GB): Bits 29:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		

Table 22.19: TDH.MEM.PAGE.PROMOTE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	If TDH.MEM.PAGE.PROMOTE succeeded, RCX returns the HPA of the removed SEPT page. Else, in RCX returns extended error information part 1. In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.PAGE.PROMOTE merges 512 private pages, which are consecutive both in the HPA space and in the GPA space. It removes the Secure EPT leaf page that formerly mapped those pages.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.20: TDH.MEM.PAGE.PROMOTE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	Removed Secure EPT page	SEPT_PAGE	R	Private	$2^{12+9*Level}$ Bytes	Exclusive	None	None
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	HPA	Merged HPA range	Blob	None	Private	N/A	Exclusive	None	None
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT Tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.PAGE.PROMOTE checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified merged page level is either 1 (2MB) or 2 (1GB) – see 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and locate the Secure EPT parent entry of the GPA range to be promoted to a merged large page.
7. Check the Secure EPT entry:
   7.1. It must be a non-leaf entry.
   7.2. It must be blocked (SEPT_BLOCKED) .
8. Get the HPA of the Secure EPT page, which currently maps the GPA range to be promoted, from the Secure EPT above. Get its PAMT entry.
9. Check that TLB tracking has been done, based on the above Secure EPT page's PAMT.BEPOCH.
10. Scan the content of the above Secure EPT page and check all 512 entries:
    10.1. They are leaf entries (this also implies that the corresponding pages are PT_REG).
    10.2. Their state is SEPT_PRESENT.
    10.3. Have contiguous HPA mapping aligned to the promoted range size.

If successful, the above checks imply that:

- The 2MB or 1GB GPA range to be promoted has a corresponding single HPA range and a single PAMT entry (PAMT_2M or PAMT_1G, respectively) owned by the current guest TD, and its current PAMT.PT is PAMT_NDA.
- The 512 child PAMT entries (PAMT_2M or PAMT_4K, respectively) of the above are owned by the current guest TD, and their PAMT.PT is PAMT_REG.

The function then does the following:

11. Merge the corresponding 512 physical pages into a single larger physical page:
    11.1. Set the small page (PAMT_4K or PAMT_2M) entries state to PT_NDA.
    11.2. Set the parent (PAMT_2M or PAMT_1G respectively) entry to PT_REG.
12. Atomically set the promoted Secure EPT entry to SEPT_PRESENT leaf entry pointing to the merged HPA range.

13. Remove the Secure EPT page that previously mapped the 512 physical pages:
    13.1. Atomically decrement TDR.CHLDCNT by 1.
        13.1.1. Note that CHLDCNT counts the number of 4KB pages. The change is due only to the removal of the Secure EPT page.
    13.2. Update the PAMT entry of the removed Secure EPT page to PT_NDA.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.21: TDH.MEM.PAGE.PROMOTE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_LEAF	
TDX_EPT_INVALID_PROMOTE_CONDITIONS	
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_NOT_BLOCKED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.PAGE.PROMOTE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TLB_TRACKING_NOT_DONE	

22.2.6. TDH.MEM.PAGE.RELOCATE Leaf

Relocate a 4KB mapped page from its current host physical address to another.

Table 22.22: TDH.MEM.PAGE.RELOCATE Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that maps the private page to be relocated, must be 0 (i.e., 4KB) (see 20.5.1).
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address of the private page to be relocated
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		
R8	Host physical address of the relocated page target (HKID bits must be 0)		

Table 22.23: TDH.MEM.PAGE.RELOCATE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.PAGE.RELOCATE replaces a mapped 4KB page mapping target HPA by moving the current page content to a new target HPA and updating the Secure-EPT mapping to the new target HPA. On successful operation, the previous mapped HPA target is marked is free in the PAMT.

To understand the table and text below, please refer to Ch. 17, which explains the general aspects of the Intel TDX API.

Table 22.24: TDH.MEM.PAGE.RELOCATE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	TD private page	Blob	R	Private	4KB	Exclusive	None	None
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Explicit	R8	HPA	Target physical page	Blob	RW	Private	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.PAGE.RELOCATE checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The target page metadata in PAMT must be correct (PT must be PT_NDA).

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand and level and find the currently mapped HPA.
7. Check the Secure EPT entry is a blocked (SEPT_BLOCKED or SEPT_PENDING_BLOCKED) leaf entry.
8. Check that the currently mapped HPA is different than the target HPA.
9. Check that TLB tracking was done.

If successful, the function does the following:

10. If the page state is SEPT_BLOCKED, copy the currently mapped page content to the target page, using the TD's ephemeral private HKID and direct writes (MOVDIR64B).
11. Free the currently mapped HPA by setting its PAMT.PT to PT_NDA.
12. Update the target page's PAMT entry with the PT_REG page type and the TDR physical address as the OWNER.
13. Update the Secure EPT entry with the target page HPA. Set its state to SEPT_PRESENT or SEPT_PENDING depending on whether its previous state was SEPT_BLOCKED or SEPT_PENDING_BLOCKED, respectively.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.25: TDH.MEM.PAGE.RELOCATE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_NOT_BLOCKED	
TDX_OPERAND_ADDR_RANGE_ERROR	

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.PAGE.RELOCATE is successful
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TLB_TRACKING_NOT_DONE	

22.2.7. TDH.MEM.PAGE.REMOVE Leaf

Remove a GPA-mapped 4KB, 2MB or 1GB private page from a TD.

Table 22.26: TDH.MEM.PAGE.REMOVE Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that maps the private page to be removed: either 0 (4KB), 1 (2MB) or 2 (1GB) – see 20.5.1.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address of the private page to be removed
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		

Table 22.27: TDH.MEM.PAGE.REMOVE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	If TDH.MEM.PAGE.REMOVE succeeded, RCX returns the HPA of the removed page. In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.PAGE.REMOVE removes a 4KB, 2MB or 1GB private page from the TD's Secure EPT tree. On successful operation, it marks the physical page as free in PAMT.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.28: TDH.MEM.PAGE.REMOVE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	TD private page	Blob	R	Private	$2^{12+9*Level}$ Bytes	Exclusive	None	None
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.PAGE.REMOVE checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified level is either 0 (4KB), 1 (2MB) or 2 (1GB) – see 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and find the page to be removed.
7. Check the page's parent Secure EPT entry is a blocked leaf entry (SEPT_BLOCKED or SEPT_PENDING_BLOCKED).
8. Check that TLB tracking was done.

If successful, the function does the following:

9. Atomically decrement TDR.CHLDCNT by 1, 512 or 512^2 depending on the removed TD private page size (4KB, 2MB or 1GB, respectively).
10. Free the physical page:
    10.1. If the level is 0 (4KB), set the PAMT entry of the removed TD private page to PT_NDA.
    10.2. Else (levels 1 or 2, 2MB or 1GB respectively), set the PAMT entry of the removed TD private page to PT_NDA.
11. Set the parent Secure EPT entry to SEPT_FREE.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.29: TDH.MEM.PAGE.REMOVE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_LEAF	
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_NOT_BLOCKED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.

Completion Status Code	Description
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.PAGE.REMOVE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TLB_TRACKING_NOT_DONE	

22.2.8. TDH.MEM.RANGE.BLOCK Leaf

Block a TD private GPA range (i.e., a Secure EPT page or a TD private page) at any level (4KB, 2MB, 1GB, 512GB, 256TB, etc.) from creating new GPA-to-HPA address translations.

Table 22.30: TDH.MEM.RANGE.BLOCK Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that maps the GPA range to be blocked – see 20.5.1 Level must between 0 and 3 for a 4-level EPT or between 0 and 4 for a 5-level EPT.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the GPA range to be blocked Depending on the level, the following least significant bits must be 0: Level 0 (EPTE): None Level 1 (EPDE): Bits 20:12 Level 2 (EPDPTE): Bits 29:12 Level 3 (EPML4E): Bits 38:12 Level 4 (EPML5E): Bits 47:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		

Table 22.31: TDH.MEM.RANGE.BLOCK Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.RANGE.BLOCK finds the Secure EPT entry for the given GPA and level, and it marks it as blocked (SEPT_BLOCKED or SEPT_PENDING_BLOCKED as appropriate). It records the current TD's TLB epoch in the PAMT entry of the physical Secure EPT page or TD private page mapped by the blocked Secure EPT entry.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.32: TDH.MEM.RANGE.BLOCK Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	Secure EPT page or TD private page	Blob	None	Private	$2^{12+9*Level}$ Bytes	None	None	None
Explicit	RDX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A
Implicit	N/A	GPA	Secure EPT entry	SEPT Entry	RW	Private	N/A	Transaction	N/A	N/A

TDH.MEM.RANGE.BLOCK checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified level is of an EPT entry – i.e., 0 to 3 for 4-level EPT or 0 to 4 for 5-level EPT. See 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and find the Secure EPT entry to be blocked.
7. Check the Secure EPT entry is not free and not blocked (its state should be SEPT_PRESENT or SEPT_PENDING).

If passed:

8. Block the Secure EPT entry. Use an atomic operation (LOCK CMPXCHG) to check that the Secure EPT entry has not change and to set its state to SEPT_BLOCKED (if it was SEPT_PRESENT) or SEPT_PENDING_BLOCKED (if it was SEPT_PENDING).

If passed:

9. Read the TD's epoch (TDCS.TD_EPOCH), and write it to the PAMT entry of the blocked Secure EPT page or TD private page (PAMT.BEPOCH).

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.33: TDH.MEM.RANGE.BLOCK Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_FREE	
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_ALREADY_BLOCKED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.RANGE.BLOCK is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.9. TDH.MEM.RANGE.UNBLOCK Leaf

Remove the blocking of a TD private GPA range (i.e., a Secure EPT page or a TD private page), at any level (4KB, 2MB, 1GB, 512GB, 256TB etc.) previously blocked by TDH.MEM.RANGE.BLOCK.

Table 22.34: TDH.MEM.RANGE.UNBLOCK Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that maps the GPA range to be unblocked – see 20.5.1 Level must between 0 and 3 for a 4-level EPT or between 0 and 4 for a 5-level EPT.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address range to be unblocked Depending on the level, the following least significant bits must be 0: Level 0 (EPTE): None Level 1 (EPDE): Bits 20:12 Level 2 (EPDPTE): Bits 29:12 Level 3 (EPML4E): Bits 38:12 Level 4 (EPML5E): Bits 47:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		

Table 22.35: TDH.MEM.RANGE.UNBLOCK Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.RANGE.UNBLOCK finds the blocked Secure EPT entry for the given GPA and level. It checks that the entry has been blocked and TLB tracking has been done, and then it marks the entry as non-blocked (SEPT_PRESENT or SEPT_PENDING as appropriate).

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.36: TDH.MEM.RANGE.UNBLOCK Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	Secure EPT page or TD private page	Blob	None	Private	$2^{12+9*Level}$ Bytes	None	None	None
Explicit	RDX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.RANGE.UNBLOCK checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified level is of an EPT entry (i.e., 0 to 3 for 4-level EPT or 0 to 4 for 5-level EPT) – see 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and find the Secure EPT page or TD private page to be unblocked.
7. Check the page's parent Secure EPT entry is blocked (SEPT_BLOCKED or SEPT_PENDING_BLOCKED).
8. Check that TLB tracking was done.

If successful, the function does the following:

9. Unblock the Secure EPT entry. Atomically set its state to SEPT_PRESENT (if it was SEPT_BLOCKED) or SEPT_PENDING (if it was SEPT_PENDING_BLOCKED).

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.37: TDH.MEM.RANGE.UNBLOCK Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_NOT_BLOCKED	

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.RANGE.UNBLOCK is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TLB_TRACKING_NOT_DONE	

22.2.10. TDH.MEM.RD Leaf

Read a 64b chunk from a debuggable guest TD private memory.

Table 22.38: TDH.MEM.RD Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The guest physical address of a naturally aligned 8-byte chunk of a guest TD private page
RDX	Host physical address of the parent TDR page (HKID bits must be 0)

Table 22.39: TDH.MEM.RD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
R8	Content of the memory chunk In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.RD reads a 64b chunk from a debuggable guest TD private memory.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.40: TDH.MEM.RD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA	TD private memory	Blob	R	Private	8B	None	None	None
Explicit	RDX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Shared	N/A	N/A

TDH.MEM.RD checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD is debuggable (TDCS.ATTRIBUTES.DEBUG is 1).

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand and find the leaf entry.
7. Check that the Secure EPT entry state is PRESENT.

If passed:

8. Read the content of the memory chunk.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.41: TDH.MEM.RD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_PRESENT	
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NON_DEBUG	

Completion Status Code	Description
TDX_TD_NOT_INITIALIZED	

22.2.11. TDH.MEM.SEPT.ADD Leaf

Add and map a 4KB Secure EPT page to a TD.

Table 22.42: TDH.MEM.SEPT.ADD Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the non-leaf Secure EPT entry that will map the new Secure EPT page – see 20.5.1 Level must between 1 and 3 for a 4-level EPT or between 1 and 4 for a 5-level EPT.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address of to be mapped for the new Secure EPT page Depending on the level, the following least significant bits must be 0: Level 1 (EPT): Bits 20:12 Level 2 (EPD): Bits 29:12 Level 3 (EPDPT): Bits 38:12 Level 4 (EPML4): Bits 47:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		
R8	Host physical address of the new Secure EPT page to be added to the TD (HKID bits must be 0)		
Other	Unmodified		

Table 22.43: TDH.MEM.SEPT.ADD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Secure EPT entry architectural content – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. • In case of successful operation, the requested entry's architectural content is returned. • In case of EPT walk error, the architectural content of the Secure EPT entry where the error was detected is returned. In other cases, RCX returns 0.
RDX	Secure EPT entry level and state – see 20.5.2 • In case of successful operation, the requested entry's information is returned. • In case of EPT walk error, the information of the Secure EPT entry where the error was detected is returned. In other cases, RDX returns 0.

Operand	Description
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.SEPT.ADD adds a 4KB Secure EPT page to a TD and maps it to the provided GPA. It initializes the page to hold 512 free entries using the TD's ephemeral private key.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.44: TDH.MEM.SEPT.ADD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	Secure EPT page (GPA)[11]	SEPT_PAGE	RW	Private	$2^{12+9*Level}$ Bytes	N/A	N/A	N/A
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Explicit	R8	HPA	Secure EPT page (HPA)[11]	SEPT_PAGE	RW	Private	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.SEPT.ADD checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified level is of an EPT non-leaf entry – i.e., 1 to 3 for 4-level EPT or 1 to 4 for 5-level EPT. See 20.5.1 for a definition of EPT level.
6. The target page metadata in PAMT must be correct (PT must be PT_NDA).

If successful, the function does the following:

7. Walk the Secure EPT based on the GPA operand, and find the parent EPT entry for the new Secure EPT page.

If the Secure EPT entry is marked as SEPT_FREE:

8. Initialize the new Secure EPT page to 0, indicating 512 entries in the SEPT_FREE state, using the TD's ephemeral private HKID and direct writes (MOVDIR64B).
9. Update the parent Secure EPT entry with the new Secure EPT page HPA and SEPT_PRESENT state.
10. Increment TDR.CHLDCNT.

[11] RCX and R8 denote the same Secure EPT page operand, using HPA and GPA respectively

11. Update the new Secure EPT page's PAMT entry with the PT_EPT page type and the TDR physical address as the OWNER.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.45: TDH.MEM.SEPT.ADD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_FREE	
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.SEPT.ADD is successful
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.12. TDH.MEM.SEPT.RD Leaf

Read a Secure EPT entry.

Table 22.46: TDH.MEM.SEPT.RD Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry to read – see 20.5.1 Level must between 0 and 3 for a 4-level EPT or between 0 and 4 for a 5-level EPT.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address for the Secure EPT entry to read Depending on the level, the following least significant bits must be 0: Level 0 (EPTE): None Level 1 (EPDE): Bits 20:12 Level 2 (EPDPTE): Bits 29:12 Level 3 (EPML4E): Bits 38:12 Level 4 (EPML5E): Bits 47:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		

Table 22.47: TDH.MEM.SEPT.RD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Secure EPT entry architectural content – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. • In case of successful operation, the requested entry's architectural content is returned. • In case of EPT walk error, the architectural content of the Secure EPT entry where the error was detected is returned. • In other cases, RCX returns 0.
RDX	Secure EPT entry level and state – see 20.5.2 • In case of successful operation, the requested entry's information is returned. • In case of EPT walk error, the information of the Secure EPT entry where the error was detected is returned. • In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.SEPT.RD reads a Secure EPT entry.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.48: TDH.MEM.SEPT.RDSEPT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	Secure EPT entry	SEPT_ENTRY	R	Private	$2^{12+9*Level}$ Bytes	None	None	None
Explicit	RDX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT Tree	N/A	R	Private	N/A	Shared	N/A	N/A

TDH.MEM.SEPT.RD checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified level is of an EPT entry (i.e., 0 to 3 for 4-level EPT or 0 to 4 for 5-level EPT) – see 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and find the Secure EPT entry.
7. Translate the internal Secure EPT content into an architectural representation (returned in RCX) and TDX state and level (returned in RDX).

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.49: TDH.MEM.SEPT.RD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	

Completion Status Code	Description
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.SEPT.RD is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.13. TDH.MEM.SEPT.REMOVE Leaf

Remove an empty 4KB Secure EPT page from a TD.

Table 22.50: TDH.MEM.SEPT.REMOVE Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the non-leaf Secure EPT entry that maps the Secure EPT page to be removed – see 20.5.1 Level must be between 1 and 3 for a 4-level EPT or between 1 and 4 for a 5-level EPT.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address for the Secure EPT page to be removed Depending on the level, the following least significant bits must be 0: Level 1 (EPT): Bits 20:12 Level 2 (EPD): Bits 29:12 Level 3 (EPDPT): Bits 38:12 Level 4 (EPML4): Bits 47:12
	63:52	Reserved	Reserved: must be 0
RDX	Host physical address of the parent TDR page (HKID bits must be 0)		

Table 22.51: TDH.MEM.SEPT.REMOVE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	If TDH.MEM.SEPT.REMOVE succeeded, RCX returns the HPA of the removed SEPT page. Else, in RCX returns extended error information part 1. In case of EPT walk error, Secure EPT entry architectural content where the error was detected – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, Secure EPT entry level and state where the error was detected – see 20.5.2 In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.SEPT.REMOVE removes an empty Secure EPT page, with all 512 marked as SEPT_FREE, from the TD's Secure EPT tree. On successful operation, it marks the 4KB physical page as free in PAMT.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.52: TDH.MEM.SEPT.REMOVE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA and Level	Secure EPT page	SEPT_PAGE	R	Private	$2^{12+9*Level}$ Bytes	Exclusive	None	None
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT Tree	N/A	RW	Private	N/A	Exclusive	N/A	N/A

TDH.MEM.SEPT.REMOVE checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The specified level is of a non-leaf EPT entry (i.e., 1 to 3 for 4-level EPT or 1 to 4 for 5-level EPT) – see 20.5.1 for a definition of EPT level.

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand, and find the Secure EPT page to be removed.
7. Check the page's parent Secure EPT entry is a blocked (SEPT_BLOCKED) non-leaf entry.
8. Check that TLB tracking was done.
9. Scan the Secure EPT page content and check all 512 entries are SEPT_FREE.

If successful, the function does the following:

10. Atomically decrement TDR.CHLDCNT.
11. Set the PAMT entry of the removed Secure EPT page to PT_NDA.
12. Set the parent Secure EPT entry to SEPT_FREE.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.53: TDH.MEM.SEPT.REMOVE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_LEAF	

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_FREE	
TDX_EPT_WALK_FAILED	
TDX_GPA_RANGE_NOT_BLOCKED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MEM.SEPT.REMOVE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TLB_TRACKING_NOT_DONE	

22.2.14. TDH.MEM.TRACK Leaf

Increment the TD's TLB epoch counter.

Table 22.54: TDH.MEM.TRACK Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of the parent TDR page (HKID bits must be 0)

Table 22.55: TDH.MEM.TRACK Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.TRACK increments the TD's TLB epoch counter.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.56: TDH.MEM.TRACK Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDR	RW	Opaque	4KB	Shared(i)	N/A	N/A
Implicit	N/A	N/A	TDCS Epoch Tracking Fields	N/A	RW	Opaque	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).

If successful, the function does the following as a critical section, protected by exclusively locking the TDCS epoch tracking fields TD_EPOCH and REFCOUNT. A concurrent TDH.VP.ENTER may cause this locking to fail with a TDX_OPERAND_BUSY status code; in this case the caller is expected to retry TDH.MEM.TRACK.

5. Lock the TDCS epoch tracking fields in exclusive mode.
6. Check that the TD's previous epoch's REFCOUNT is 0. This helps ensure that no REFCOUNT information will be lost when TD_EPOCH is incremented in the next step.
7. If successful, increment the TD's epoch counter (TDCS.TD_EPOCH).
8. Release the exclusive mode locking of the epoch tracking fields.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.57: TDH.MEM.TRACK Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation. Note the special case where the indicated operand is TLB_EPOCH. This may happen due to a conflict with TDH.VP.ENTER. The host VMM should retry TDH.MEM.TRACK.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_PREVIOUS_TLB_EPOCH_BUSY	
TDX_SUCCESS	TDH.MEM.TRACK is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.15. TDH.MEM.WR Leaf

Write a 64b chunk from a debuggable guest TD private memory.

Table 22.58: TDH.MEM.WR Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The guest physical address of a naturally aligned 8-byte chunk of a guest TD private page
RDX	Host physical address of the parent TDR page (HKID bits must be 0)
R8	Data to be written to memory

Table 22.59: TDH.MEM.WR Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Secure EPT entry architectural content – see 20.5.2 The architectural content represents how the Secure EPT maps a private memory page or a Secure EPT page, and may be different than the actual contents of the Secure EPT entry. Software should consult the Secure EPT information returned in RDX. • In case of successful operation, the requested entry's architectural content is returned. • In case of EPT walk error, the architectural content of the Secure EPT entry where the error was detected is returned. In other cases, RCX returns 0.
RDX	Secure EPT entry level and state – see 20.5.2 • In case of successful operation, the requested entry's information is returned. • In case of EPT walk error, the information of the Secure EPT entry where the error was detected is returned. In other cases, RDX returns 0.
R8	Previous content of the memory chunk In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MEM.WR writes a 64b chunk to a debuggable guest TD private memory.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.60: TDH.MEM.RD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA	TD private memory	Blob	RW	Private	8B	None	None	None
Explicit	RDX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	RW	Private	N/A	Shared	N/A	N/A

TDH.MEM.WR checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD is debuggable (TDCS.ATTRIBUTES.DEBUG is 1).

If successful, the function does the following:

6. Walk the Secure EPT based on the GPA operand and find the leaf entry.
7. Check that the Secure EPT entry state is PRESENT.

If passed:

8. Read the content of the memory chunk.
9. Write the new content of the memory chunk.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.61: TDH.MEM.WR Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_PRESENT	
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	

Completion Status Code	Description
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NON_DEBUG	
TDX_TD_NOT_INITIALIZED	

22.2.16. TDH.MNG.ADDCX Leaf

Add a TDCX page to a guest TD.

Table 22.62: TDH.MNG.ADDCX Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a page where TDCX will be added (HKID bits must be 0)
RDX	The physical address of the owner TDR page (HKID bits must be 0)

Table 22.63: TDH.MNG.ADDCX Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.ADDCX adds a TDCX page, which is a child of the specified TDR.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX Interface functions.

Table 22.64: TDH.MNG.ADDCX Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDCX page	Blob	RW	Opaque	4KB	Exclusive	Shared	Shared
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Exclusive	Shared	Shared

In addition to the explicit memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD must not have been initialized (TDR.INIT is FALSE).
4. The number of TDCX pages (TDR.NUM_TDCX) is smaller than the required number.
5. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
6. The new TDCX page metadata in PAMT must be correct (PT must be PT_NDA).

If successful, the function does the following:

7. Initialize the TDCX page contents using direct writes (MOVDIR64B).
8. Set the TDCX pointer entry in the TDR.TDCX_PA array.
9. Increment TDR.NUM_TDCX.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.65: TDH.MNG.ADDCX Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.ADDCX is successful
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_INITIALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TDCX_NUM_INCORRECT	

22.2.17. TDH.MNG.CREATE Leaf

Create a new guest TD and its TDR root page.

Table 22.66: TDH.MNG.CREATE Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	The physical address of a page where TDR will be created (HKID bits must be 0)		
RDX	**Bits**	**Name**	**Description**
	15:0	HKID	The TD's ephemeral private HKID
	63:16	Reserved	Reserved: must be 0

Table 22.67: TDH.MNG.CREATE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.CREATE creates a TDR page which is the root page of a new guest TD.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.68: TDH.MNG.CREATE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	KOT	KOT	N/A	Hidden	N/A	Exclusive	N/A	N/A

In addition to the explicit memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_NDA).
2. The value of the specified HKID must be in the range configured for TDX.
3. The KOT entry for the specified HKID must be marked as HKID_FREE.

If successful, the function does the following:

4. Zero out the TDR page contents using direct write (MOVDIR64B).
5. Initialize the key management fields.
6. Initialize the state variables.
7. Initialize the TD management fields.
8. Mark the KOT entry for the specified HKID as HKID_ASSIGNED.
9. Initialize the TDR page metadata in PAMT.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.69: TDH.MNG.CREATE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_HKID_NOT_FREE	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.CREATE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	

22.2.18. TDH.MNG.INIT Leaf

Initialize TD-scope control structures TDR and TDCS.

Table 22.70: TDH.MNG.INIT Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)
RDX	The physical address (including HKID bits) of an input TD_PARAMS_STRUCT

Table 22.71: TDH.MNG.INIT Output Operands Definition

Operand	Description		
RAX	SEAMCALL instruction return code – see 22.2.1		
RCX	Extended error information In case of a TD_PARAMS_STRUCT.CPUID_CONFIG error, RCX returns the applicable CPUID information as shown below. In all other cases, RCX returns 0.		
	Bits	**Name**	**Description**
	31:0	LEAF	CPUID leaf number
	63:32	SUBLEAF	CPUID sub-leaf number: if sub-leaf is not applicable, value is -1 (0xFFFFFFFF).
Other	Unmodified		

Leaf Function Latency

TDH.MNG.INIT execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.INIT initializes the TD-scope control structures TDR and TDCS based on a set of TD parameters provided as input.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.72: TDH.MNG.INIT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	RW	Opaque	4KB	Exclusive	Shared	Shared
Explicit	RDX	HPA	TD Parameters	TD_PARAMS	R	Shared	1024B	None	N/A	N/A
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Exclusive(i)	N/A	N/A

In addition to the explicit memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD must not have been initialized (TDR.INIT is FALSE).
4. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
5. All the TDCX pages have been added (by TDH.MNG.ADDCX).

If successful, the function does the following:

6. Set the TDCS TD management fields to their initial values.
7. Read the input parameters structure fields.
8. Check the input parameters and initialize the TDCS logical structure.
   8.1. Check that ATTRIBUTES and XFAM bits that must be fixed-0 or fixed-1 are set correctly.
   8.2. Check XFAM bit groups that must have certain values (e.g., AVX bits 7:5).

If passed:

9. Initialize EPTP to point to TDCS.SEPT_ROOT.
10. Initialize the MSR bitmaps based on ATTRIBUTES and XFAM.
11. Initialize the TDCS measurement fields.
12. Mark the TD as initialized (set TDR.INIT to TRUE).

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.73: TDH.MNG.INIT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_BOOT_NT4_SET	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.INIT is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_INITIALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TDCX_NUM_INCORRECT	

22.2.19. TDH.MNG.KEY.CONFIG Leaf

Configure the TD ephemeral private key on a single package.

Table 22.74: TDH.MNG.KEY.CONFIG Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)

Table 22.75: TDH.MNG.KEY.CONFIG Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Latency

TDH.MNG.KEY.CONFIG execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.KEY.CONFIG configures the TD's ephemeral private key on a single package.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.76: TDH.MNG.KEY.CONFIG Operands Information

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	KETs on current package	N/A	N/A	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. HKID has been assigned to the TD; TDR.LIFECYCLE_STATE is TD_HKID_ASSIGNED.
4. The key has not yet been configured by TDH.MNG.KEY.CONFIG on this package.

If successful, the function does the following:

5. Configure the TD ephemeral private key on the package.
   5.1. This operation may fail due to a conflict with a concurrent TDH.MNG.KEY.CONFIG or PCONFIG running on the same package.
   5.2. A CPU-generated random key is used. The operation may fail due to lack of entropy.
6. If the key has been configured on all the packages, set TDR.LIFECYCLE_STATE to TD_KEYS_CONFIGURED.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.77: TDH.MNG.KEY.CONFIG Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_KEY_CONFIGURED	
TDX_KEY_GENERATION_FAILED	Failed to generate a random key. This is typically caused by an entropy error of the CPU's random number generator, and may be impacted by RDSEED, RDRAND or PCONFIG executing on other LPs. The operation should be retried.
TDX_LIFECYCLE_STATE_INCORRECT	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation. Specifically, key configuration may fail due to a concurrently running PCONFIG instruction.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.KEY.CONFIG is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	

22.2.20. TDH.MNG.KEY.FREEID Leaf

End the platform cache flush sequence, and mark applicable HKIDs in KOT as free.

Table 22.78: TDH.MNG.KEY.FREEID Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)

Table 22.79: TDH.MNG.KEY.FREEID Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.KEY.FREEID ends the platform cache flush sequence for the HKIDs associated with the specified TD after TDH.PHYMEM.CACHE.WB has been executed on all the required packages. It marks the TD's HKIDs in KOT as free, and the TD itself as being torn down.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.80: TDH.MNG.KEY.FREEID Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	KOT	KOT	N/A	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. TLB and VMCS caches associated with the HKID have been flushed, and no memory associated with this HKID may be accessed:
   2.1. TDR.LIFECYCLE_STATE is TD_BLOCKED.
   2.2. The KOT entry for the TD's private HKID is marked as HKID_FLUSHED.
   2.3. The KOT entry for the TD's private HKID indicates that TDH.PHYMEM.CACHE.WB has been executed on all applicable packages or cores.

If successful, the function does the following:

3. Mark the KOT entry as HKID_FREE.
4. Set TDR.LIFECYCLE_STATE to TD_TEARDOWN.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.81: TDH.MNG.KEY.FREEID Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_LIFECYCLE_STATE_INCORRECT	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.KEY.FREEID is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_WBCACHE_NOT_COMPLETE	

22.2.21. TDH.MNG.KEY.RECLAIMID Leaf

This function is provided for backward compatibility.

Table 22.82: TDH.MNG.KEY.RECLAIMID Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)

Table 22.83: TDH.MNG.KEY.RECLAIMID Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.KEY.RECLAIMID is provided for backward compatibility. It does not do anything except returning a constant TDX_SUCCESS status.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.84: TDH.MNG.KEY.RECLAIMID Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_SUCCESS	TDH.MNG.KEY.RECLAIMID is successful.

22.2.22. TDH.MNG.RD Leaf

Read a TD-scope control structure field of a TD.

Table 22.85: TDH.MNG.RD Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)
RDX	Field code

Table 22.86: TDH.MNG.RD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
R8	Contents of the field In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.RD reads a TD-scope control structure field of a TD.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.87: TDH.MNG.RD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	R	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The provided field code is valid and indicates a readable field per the TD's debug attribute (TDCS.ATTRIBUTES.DEBUG).

If the above checks pass, the function does the following:

6. Read the TD field.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.88: TDH.MNG.RD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FIELD_NOT_READABLE	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.RD is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.23. TDH.MNG.VPFLUSHDONE Leaf

Check that none of the TD's VCPUs are associated with an LP.

Table 22.89: TDH.MNG.VPFLUSHDONE Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)

Table 22.90: TDH.MNG.VPFLUSHDONE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.VPFLUSHDONE checks that none of the TD's VCPUs are associated with an LP, and it then prepares for cache flushing by TDH.PHYMEM.CACHE.WB.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.91: TDH.MNG.VPFLUSHDONE Operands Information

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Exclusive(i)	N/A	N/A
Implicit	N/A	N/A	KOT	KOT	N/A	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. TDR.LIFECYCLE_STATE is either TD_HKID_ASSIGNED or TD_KEYS_CONFIGURED.
3. The KOT entry for the TD's assigned HKID in the list must be marked as HKID_ASSIGNED.
4. None of the TD's VCPUs are associated with an LP (either the TD has not been initialized by TDH.MNG.INIT, or TDCS.NUM_ASSOC_VCPUS is 0).

If successful, the function does the following:

5. Set a bitmap in the KOT entry to track the required subsequent TDH.PHYMEM.CACHE.WB operations.
6. Set TDR.LIFECYCLE_STATE to TD_BLOCKED.
7. Mark the KOT entry as HKID_FLUSHED.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.92: TDH.MNG.VPFLUSHDONE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FLUSHVP_NOT_DONE	
TDX_LIFECYCLE_STATE_INCORRECT	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.VPFLUSHDONE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	

22.2.24. TDH.MNG.WR Leaf

Write a TD-scope control structure field of a TD.

Table 22.93: TDH.MNG.WR Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDR page (HKID bits must be 0)
RDX	Field code
R8	Data to write to the field
R9	64b mask to indicate which bits of the value in R8 are to be written to the field

Table 22.94: TDH.MNG.WR Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
R8	Previous content of the field In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MNG.WR writes a TD-scope control structure field of a TD. The specific bits of the value (R8) are written as specified by the write mask (R9). Writing is subject to the field's writability.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.95: TDH.MNG.WR Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR)
2. The TD is not in a FATAL state (TDR.FATAL is FALSE)
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED)
4. TDCS must have been initialized (TDR.INIT is TRUE)
5. The provided field code is valid and indicates a writable field per the TD's debug attribute (TDCS.ATTRIBUTES.DEBUG).

If the above checks pass, the function does the following:

6. Derive the field attributes (read mask, write mask, VMCS field code or offset in TDVPS) from the field code provided in RDX.

If passed:

7. Derive the effective write mask by bitwise-anding the write mask derived above with the write mask provided in R9. If the effective write mask is 0, then fail; the field in not writable.

If passed:

8. Read the field value.
9. Update the field value from the input value in R8, per the effective write mask.

If passed:

10. Mask out the previous field value with the read mask derived earlier, and return in R8.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.96: TDH.MNG.WR Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FIELD_NOT_READABLE	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MNG.WR is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.25. TDH.MR.EXTEND Leaf

Extend the MRTD measurement register in the TDCS with the measurement of the indicated chunk of a TD page.

Table 22.97: TDH.MR.EXTEND Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The GPA of the TD page chunk to be measured
RDX	The TDR page of the target TD (HKID bits must be 0)

Table 22.98: TDH.MR.EXTEND Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RCX	Extended error information part 1 In case of EPT walk error, Secure EPT entry where the error was detected In other cases, RCX returns 0.
RDX	Extended error information part 2 In case of EPT walk error, EPT level where the error was detected In other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MR.EXTEND updates the MRTD measurement register in the TDCS with the measurement of the indicated chunk of a TD private page. For pages whose contents need to be measured, once the page is copied into the TD memory area, the host VMM will call TDH.MR.EXTEND multiple times to measure the pages contents into MRTD. TDEXEND can be executed only before TDH.MR.FINALIZE.

Note: TDH.MR.EXTEND works on a 256B chunk of a page, not on a full page, due to instruction latency considerations.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.99: TDH.MR.EXTEND Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	GPA	TD private page chunk	Blob	R	Private	256B	None	None	None
Explicit	RDX	HPA	TDR page	TDR	R	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	4KB	Exclusive(i)	N/A	N/A
Implicit	N/A	GPA	Secure EPT tree	N/A	R	Private	N/A	Shared	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD build and measurement must not have been finalized (by TDH.MR.FINALIZE).
6. The page must be mapped and accessible in the Secure EPT.

If successful, the function does the following:

7. Update the TD measurement in TDCS based on the chunk's GPA and contents.
8. Extend TDCS.MRTD with the chunk's GPA and contents. Extension is done using SHA384, with three 128B extension buffers. The first extension buffer is composed as follows:
   - Bytes 0 through 8 contain the ASCII string "MR.EXTEND".
   - Bytes 16 through 23 contain the GPA (in little-endian format).
   - All the other bytes contain 0.

   The other two extension buffers contain the chunk's contents.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.100: TDH.MR.EXTEND Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_EPT_ENTRY_NOT_PRESENT	
TDX_EPT_WALK_FAILED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MR.EXTEND is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_FINALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.26. TDH.MR.FINALIZE Leaf

TDH.MR.FINALIZE completes measurement of the initial TD contents and marks the TD as ready to run.

Table 22.101: TDH.MR.FINALIZE Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of the parent TDR page (HKID bits must be 0)

Table 22.102: TDH.MR.FINALIZE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.MR.FINALIZE completes the measurement of the initial TD contents and marks the TD as finalized.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.103: TDH.MR.FINALIZE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDR page	TDR	R	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	4KB	Exclusive(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD build and measurement must not have been finalized (by TDH.MR.FINALIZE).

If successful, the function does the following:

6. Finalize the TD measurement.
7. Mark the TD as finalized.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.104: TDH.MR.FINALIZE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.MR.FINALIZE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_FINALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.27. TDH.PHYMEM.CACHE.WB Leaf

TDH.PHYMEM.CACHE.WB is an interruptible and restartable function to write back the cache hierarchy on a package or a core.

Table 22.105: TDH.PHYMEM.CACHE.WB Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	Command, as described below:		
	Value	**Name**	**Description**
	0	WB_START_CMD	Start a new TDH.PHYMEM.CACHE.WB cycle with no cache invalidation.
	1	WB_RESUME_CMD	Resume a previously interrupted TDH.PHYMEM.CACHE.WB cycle with no cache invalidation.
	Other		Reserved

Table 22.106: TDH.PHYMEM.CACHE.WB Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.PHYMEM.CACHE.WB writes back the cache hierarchy to memory and updates the KOT state to allow reuse of HKIDs.

- TDH.PHYMEM.CACHE.WB does not invalidate cache lines.
- The function is interruptible by external events and is restartable. In case it is interrupted by an external event, information is stored in an Intel TDX module internal table which allows the instruction to be restarted.
- The function operates on cache lines associated with any HKID.
- The function is designed to ensure write back of at least those cache lines where the state of that HKID (in the KOT) was HKID_FLUSHED at the time of the first invocation (RCX == TDH.PHYMEM.CACHE.WB_START_CMD (0)).
- Depending on the implementation, the instruction may write back additional cache lines.
- The scope at which TDH.PHYMEM.CACHE.WB operates (e.g., package or core) is determined at Intel TDX module initialization time.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.107: TDH.PHYMEM.CACHE.WB (Implicit) Operands Information

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	KOT	KOT	N/A	Hidden	N/A	Shared	N/A	N/A
Implicit	N/A	N/A	WBT entry for current scope	WBT_ENTRY	N/A	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The command value is one of the supported ones.
2. If the command is to start a new TDH.PHYMEM.CACHE.WB cycle (RCX == 0), then:
   2.1. Clear the internally saved interruption state.
   2.2. Scan the KOT: mark those HKIDs whose state is HKID_FLUSHED in an internal table; only those HKIDs will be later marked as written back and invalidated upon successful completion of TDH.PHYMEM.CACHE.WB.
   2.3. If none of the KOT entries for the requested set of HKIDs (either single or all) is in HKID_FLUSHED state, then abort with an informational code (it achieved its goal: write back and invalidate at least the HKIDs that are in the HKID_FLUSHED state).
3. Run cache write back operation on the cache hierarchy of the current package or core. This operation is long and may be interrupted by external events.
   3.1. If a previous TDH.PHYMEM.CACHE.WB has been interrupted, the operation resumes from the interruption point which has been recorded.
   3.2. In case of interruption, the current point in the write back and invalidation flow and the current HKID are recorded.
4. If the operation has not been interrupted, update the KOT as follows:
   4.1. For each KOT entry, if the entry was marked as HKID_FLUSHED at the start of the TDH.PHYMEM.CACHE.WB cycle as discussed above, use the KOT entry's bitmap to indicate that TDH.PHYMEM.CACHE.WB has been executed on this package or core.

Error and Informational Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.108: TDH.PHYMEM.CACHE.WB Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_INTERRUPTED_RESUMABLE	TDH.PHYMEM.CACHE.WB was interrupted; it is recommended to resume it with RCX indicating WB_RESUME_CMD
TDX_NO_HKID_READY_TO_WBCACHE	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDH.PHYMEM.CACHE.WB is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	

22.2.28. TDH.PHYMEM.PAGE.RDMD Leaf

Read the metadata of a page (or the metadata of the containing large page) in TDMR.

Table 22.109: TDH.PHYMEM.PAGE.RDMD Operands

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	A physical address of a 4KB page in TDMR (HKID bits must be 0)

Table 22.110: TDH.PHYMEM.PAGE.RDMD Output Operands Definition

Operand	Description		
RAX	SEAMCALL instruction return code – see 22.2.1		
RCX	Page Type (PT):		
	Value	**Name**	**Description**
	0	PT_NDA	The physical page is **Not Directly Assigned** to the Intel TDX module.
	1	PT_RSVD	The physical page is reserved for non-TDX usage.
	3	PT_REG	The physical page holds TD private memory.
	4	PT_TDR	The physical page holds the TD Root (TDR) control structure.
	8:5		The physical page holds a TD control structure.
	Other		Reserved
	In case of an error, as indicated by RAX, RCX returns 0		
RDX	OWNER: the HPA of the TD's TDR control structure page (if applicable) In case of an error, RDX returns 0		
R8	**Bits**	**Name**	**Description**
	2:0	Size	Size of the containing 4KB, 2MB or 1GB page – see 20.4.1
	63:3	Reserved	Set to 0
	In case of an error, as indicated by RAX, R8 returns 0		
R9	BEPOCH In case of an error, as indicated by RAX, R9 returns 0		
R10	Reserved: set to 0		
R11	Reserved: set to 0		
Other	Unmodified		

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.PHYMEM.PAGE.RDMD finds the containing page (4KB, 2MB or 2GB) of the given page in TDMR and reads its metadata from its PAMT entry.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.111: TDH.PHYMEM.PAGE.RDMD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	Target page	Blob	None	Opaque/ Private	4KB	Shared	Shared	Shared

If the memory operand checks per the table above pass, the function does the following:

1. Do a PAMT walk, and find the containing page and its size.

If passed:

2. Read the PAMT entry.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.112: TDH.PHYMEM.PAGE.RDMD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDH.PHYMEM.PAGE.RDMD is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	

22.2.29. TDH.PHYMEM.PAGE.RECLAIM Leaf

Reclaim a physical 4KB, 2MB or 1GB TD-owned page (i.e., TD private page, Secure EPT page or a control structure page) from a TD, given its HPA.

Table 22.113: TDH.PHYMEM.PAGE.RECLAIM Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a 4KB, 2MB or 1GB page to be reclaimed (HKID bits must be 0)

Table 22.114: TDH.PHYMEM.PAGE.RECLAIM Output Operands Definition

Operand	Description		
RAX	SEAMCALL instruction return code – see 22.2.1		
RCX	Page Type (PT):		
	Value	**Name**	**Description**
	0	PT_NDA	The physical page is **Not Directly Assigned** to the Intel TDX module.
	1	PT_RSVD	The physical page is reserved for non-TDX usage.
	3	PT_REG	The physical page holds TD private memory.
	4	PT_TDR	The physical page holds the TD Root (TDR) control structure.
	8:5		The physical page holds a TD control structure.
	Other		Reserved
	In multiple error cases, as indicated by RAX, RDX returns 0. In other error cases, RDX still returns the PT information. See the completion status codes table below for details.		
RDX	OWNER: the HPA of the TD's TDR control structure page (if applicable) In multiple error cases, as indicated by RAX, RDX returns 0. In other error cases, RDX still returns the OWNER information. See the completion status codes table below for details.		
R8	**Bits**	**Name**	**Description**
	2:0	Size	Size of the containing 4KB, 2MB or 1GB page – see 20.4.1
	63:3	Reserved	Set to 0
	In multiple error cases, as indicated by RAX, RDX returns 0. In other error cases, RDX still returns the size information. See the completion status codes table below for details.		
R9	Reserved: set to 0		
R10	Reserved: set to 0		
R11	Reserved: set to 0		
Other	Unmodified		

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.PHYMEM.PAGE.RECLAIM reclaims a TD-owned physical page from the TD.

TDH.PHYMEM.PAGE.RECLAIM can reclaim pages only if the owner TD is in the TD_TEARDOWN state.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.115: TDH.PHYMEM.PAGE.RECLAIM Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	Target page	Blob	RW	Opaque/ Private	4KB, 2MB or 1GB	Exclusive	Shared	Shared
Implicit	N/A	N/A	TDR page[12]	TDR	RW	Opaque	4KB	Shared	N/A	N/A

TDH.PHYMEM.PAGE.RECLAIM checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

The function works as follows:

1. Check that the target page metadata in PAMT are correct (PT must not be PT_NDA nor PT_RSVD).
2. If the target page is not a TDR (PT is not PT_TDR):
   2.1. Get the TDR page (pointed by the target page's PAMT.OWNER).
   2.2. Check that the TD is in teardown state (TDR.LIFECYCLE_STATE is TD_TEARDOWN).
   2.3. Atomically decrement TDR.CHLDCNT.
3. Else (target page is a TDR):
   3.1. Check that the TD is in teardown state (TDR.LIFECYCLE_STATE is TD_TEARDOWN).
   3.2. Check that TDR.CHLDCNT is 0.
4. Update the PAMT entry of the reclaimed page to PT_NDA.
5. Return the page metadata (as they were before PAMT update above).

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.116: TDH.PHYMEM.PAGE.RECLAIM Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_LIFECYCLE_STATE_INCORRECT	RCX, RDX and R9 return the actual PT, OWNER and size information.
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation. If the page is not a TDR page but the owner TDR is busy, then RCX, RDX and R9 return the actual PT, OWNER and size information.
TDX_OPERAND_INVALID	If the page physical address is not aligned on its size, then RCX, RDX and R9 return the actual PT, OWNER and size information.

[12] Except when TDR is the target page

Completion Status Code	Description
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.PHYMEM.PAGE.RECLAIM is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_ASSOCIATED_PAGES_EXIST	RCX, RDX and R9 return the actual PT, OWNER and size information.

22.2.30. TDH.PHYMEM.PAGE.WBINVD Leaf

Write back and invalidate all cache lines associated with the specified memory page and HKID.

Table 22.117: TDH.PHYMEM.PAGE.WBINVD Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	Physical address (including HKID bits) of a 4KB page in TDMR

Table 22.118: TDH.PHYMEM.PAGE.WBINVD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.PHYMEM.PAGE.WBINVD performs cache write back and invalidation on all the cache lines associated with the specified page and HKID. The page must not be in use by the Intel TDX module (i.e., not assigned to a TD as a private page or a Secure EPT page), nor used as a control structure page.

It is the responsibility of the host VMM to track which HKID is associated with the target page; the function does not check it.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.119: TDH.PHYMEM.PAGE.WBINVD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	Target page	Blob	R	Private/ Opaque	4KB	Shared	Shared	Shared

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The target page must be marked in PAMT as not controlled by the Intel TDX module (PT must be PT_NDA).

If successful, the function performs the following:

2. Write back and invalidate all the cache lines for the given target HPA and HKID.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.120: TDH.PHYMEM.PAGE.WBINVD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.PHYMEM.PAGE.WBINVD is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	

22.2.31. TDH.SYS.CONFIG Leaf

Globally configure the Intel TDX module.

Table 22.121: TDH.SYS.CONFIG Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	The physical address (including HKID bits) of an array of pointers, each containing the physical address of a single TDMR_INFO entry (see 20.8.4). The pointer array must be sorted such that TDMR base addresses (TDMR_INFO.TDMR_BASE) are sorted from the lowest to the highest base address, and TDMRs do not overlap with each other.		
RDX	The number of pointers in the above buffer, between 1 and 64		
R8	**Bits**	**Name**	**Description**
	15:0	HKID	Intel TDX global private HKID value
	63:16	Reserved	Reserved: must be 0

Table 22.122: TDH.SYS.CONFIG Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.CONFIG performs global (platform-scope) configuration of the Intel TDX module. This function is intended to be executed during OS/VMM boot, and thus it has relaxed latency requirements.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.123: TDH.SYS.CONFIG Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDMR Info Pointers	Array of HPA	R	Shared	512B	None	N/A	N/A
Explicit	N/A	HPA	TDMR Info	TDMR_INFO	R	Shared	512B	None	N/A	N/A
Implicit	N/A	N/A	All Intel TDX module internal variables	N/A	RW	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. Global and LP-scope initialization has been done:
   1.1. PL.SYS_STATE is SYSINIT_DONE.
   1.2. TDH.SYS.LP.INIT has been executed on all LPs.
2. The number of TDMR_INFO entries is at least 1 and does not exceed the supported number of TDMRs.
3. Check each physical address of to TDMR_INFO; read the applicable TDMR_INFO entry; check and update the internal TDMR_TABLE with TDMR, reserved areas and PAMT setup. The order of checks is not required to be exactly the same as described below.
   - TDMRs must be sorted in an ascending base address order.
   - For each TDMR:
     - TDMR base address must be aligned on 1GB.
     - TDMR size must be greater than 0 and a whole multiple of 1GB.
     - Any address within the TDMR must comply with the platform's maximum PA, and its HKID bits must be 0.
     - For each PAMT region (1G, 2M and 4K) of each TDMR:
       - PAMT base address must comply with the alignment requirements.
       - Any address within the PAMT range must comply with the platform's maximum PA, and its HKID bits must be 0.
       - The size of each PAMT region must be large enough to contain the PAMT for its associated TDMR.
     - Reserved areas within TDMR must be sorted in an ascending offset order.
     - A null reserved area (indicated by a size of 0) may be followed only by other null reserved areas.
     - For each reserved area within TDMR:
       - Offset and size must comply with the alignment and granularity requirements.
       - Reserved areas must not overlap.
       - Reserved areas must be fully contained within their TDMR.
   - TDMRs must not overlap with other TDMRs.
   - PAMTs must not overlap with other PAMTs.
   - TDMRs' non-reserved parts and PAMTs must not overlap (PAMTs may reside within TDMR reserved areas).
   - TDMRs' non-reserved parts must be contained in convertible memory – i.e., in CMRs.
   - PAMTs must be contained in convertible memory – i.e., in CMRs.
4. Check and set the Intel TDX global private HKID. The provided HKID must be in the TDX HKID range.

If successful, the function does the following:

5. Complete the initialization of the Intel TDX module at platform scope.
6. Set PL.SYS_STATE to SYSCONFIG_DONE.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.124: TDH.SYS.CONFIG Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_INVALID_PAMT	
TDX_INVALID_RESERVED_IN_TDMR	
TDX_INVALID_TDMR	
TDX_NON_ORDERED_RESERVED_IN_TDMR	
TDX_NON_ORDERED_TDMR	
TDX_OPERAND_INVALID	
TDX_PAMT_OUTSIDE_CMRS	
TDX_PAMT_OVERLAP	
TDX_SUCCESS	TDH.SYS.CONFIG is successful.

Completion Status Code	Description
TDX_SYS_BUSY	The operation was invoked when another TDX module operation was in progress. The operation may be retried.
TDX_SYS_CONFIG_NOT_PENDING	
TDX_SYS_SHUTDOWN	
TDX_TDMR_ALREADY_INITIALIZED	
TDX_TDMR_OUTSIDE_CMRS	

22.2.32. TDH.SYS.INFO Leaf

Provide information about the Intel TDX module and the convertible memory.

Table 22.125: TDH.SYS.INFO Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address (including HKID bits) of a buffer where the output TDSYSINFO_STRUCT will be written
RDX	The number of bytes in the above buffer
R8	The physical address (including HKID bits) of a buffer where an array of CMR_INFO will be written
R9	The number of CMR_INFO entries in the above buffer

Table 22.126: TDH.SYS.INFO Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RDX	The actual number of bytes written to the above buffer In case of an error, as indicated by RAX, RDX returns 0
R9	The number of CMR_INFO entries actually written to the above buffer In case of an error, as indicated by RAX, R9 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.INFO provides information about the Intel TDX module and about the memory configuration.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.127: TDH.SYS.INFO Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDX system information structure	TDSYSINFO_STRUCT	RW	Shared	1024B	None	N/A	N/A
Explicit	R8	HPA	CMR table	CMR_INFO_ARRAY	RW	Shared	512B	None	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. Global and LP-scope initialization has been done:
   1.1. TDH.SYS.INIT has been executed.
   1.2. TDH.SYS.LP.INIT has been executed on the current LP.

2. The number of bytes provided for returning TDSYSINFO_STRUCT (in RDX) must be at least the size of that structure.
3. The number of entries provided for returning CMR_INFO_ARRAY (in R9) must be at least the number of CMRs supported by TDX.

If successful, the function does the following:

4. Write the TDSYSINFO_STRUCT, and set RDX to the actual number of bytes written.
5. Write the CMR_INFO_ARRAY based on the CMR information in SEAMCFG, and set R9 to the number of CMRs.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.128: TDH.SYS.INFO Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDH.SYS.INFO is successful.
TDX_SYS_SHUTDOWN	
TDX_SYS_LP_INIT_NOT_DONE	

22.2.33. TDH.SYS.INIT Leaf

Globally initialize the Intel TDX module.

Table 22.129: TDH.SYS.INIT Input Operands Definition

Operand	Description		
RAX	SEAMCALL instruction leaf number – see 22.2.1		
RCX	Intel TDX module attributes		
	Bits	**Name**	**Description**
	63:0	RESERVED	Reserved: must be 0

Table 22.130: TDH.SYS.INIT Output Operands Definition

Operand	Description		
RAX	SEAMCALL instruction return code – see 22.2.1		
RCX	Extended error information part 1 If RAX returns TDX_INCORRECT_CPUID_VALUE, RCX returns the applicable CPUID information as shown below. In all other cases, RCX returns 0.		
	Bits	**Name**	**Description**
	31:0	LEAF	CPUID leaf number
	63:32	SUBLEAF	CPUID sub-leaf number: if sub-leaf is not applicable, value is -1 (0xFFFFFFFF).
RDX	Extended error information part 2 If RAX returns TDX_INCORRECT_CPUID_VALUE, RDX returns the value masks as shown below. A bit value of 1 indicates a bit position that was checked against the required value. In all other cases, RDX returns 0.		
	Bits	**Name**	**Description**
	31:0	MASK_EAX	Mask of the value returned by CPUID in EAX
	63:32	MASK_EBX	Mask of the value returned by CPUID in EBX
R8	Extended error information part 3 If RAX returns TDX_INCORRECT_CPUID_VALUE, R8 returns the value masks as shown below. A bit value of 1 indicates a bit position that was checked against the required value. In all other cases, R8 returns 0.		
	Bits	**Name**	**Description**
	31:0	MASK_ECX	Mask of the value returned by CPUID in ECX
	63:32	MASK_EDX	Mask of the value returned by CPUID in EDX
R9	Extended error information part 4 If RAX returns TDX_INCORRECT_CPUID_VALUE, R9 returns the expected values as shown below. In all other cases, R9 returns 0.		
	Bits	**Name**	**Description**
	31:0	VALUE_EAX	Value expected to be returned by CPUID in EAX

Operand	Description		
	63:32	VALUE_EBX	Value expected to be returned by CPUID in EBX
R10	Extended error information part 5 If RAX returns TDX_INCORRECT_CPUID_VALUE, R10 returns the expected values as shown below. In all other cases, R10 returns 0.		
	Bits	**Name**	**Description**
	31:0	VALUE_ECX	Value expected to be returned by CPUID in ECX
	63:32	VALUE_EDX	Value expected to be returned by CPUID in EDX
Other	Unmodified		

Special Environment Requirements

If the IA32_TSX_CTRL MSR is supported by the CPU, as enumerated by IA32_ARCH_CAPABILITIES.TSX_CTRL (bit 7), then the values of its following bits must be 0:

- RTM_DISABLE (bit 0)
- TSX_CPUID_CLEAR (bit 1)

Leaf Function Latency

TDH.SYS.INIT execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.INIT performs global (platform-scope) initialization of the Intel TDX module. This function is intended to be executed during OS/VMM boot and thus it has relaxed latency requirements.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.131: TDH.SYS.INIT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	All Intel TDX module internal variables	N/A	RW	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. Check that PL.SYS_STATE is SYSINIT_PENDING.
2. Do any global Intel TDX module initializations required for running this flow.
3. Check the memory operands per the table above.
4. Check the following conditions (no specific order is implied):
   - Enumerate CPU and platform information, and check Intel TDX module compatibility. If the Intel TDX module is compatible with multiple variants of CPU and platform features, sample the current LP's features enumeration – to be later checked to be the same on all LPs by TDH.SYS.LP.INIT. Examples of compatibility checks are:
     - The CPU must support any ISA that the Intel TDX module relies upon, such as SHA-NI.
     - The CPU must support the WBINVD scope for which the Intel TDX module was built.

- Sample and check the platform configuration on the current LP – to be later checked to be the same on all LPs by TDH.SYS.LP.INIT. For example:
  - Sample SMRR and SMRR2, check they are locked and do not overlap any CMR, and store their values to be checked later on each LP.

If successful, the function does the following:

5. Complete the initialization of the Intel TDX module at platform scope.
6. Set PL.SYS_STATE to SYSINIT_DONE.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.132: TDH.SYS.INIT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_BOOT_NT4_SET	
TDX_CMR_LIST_INVALID	
TDX_CPUID_LEAF_NOT_SUPPORTED	
TDX_CPUID_LEAF_1F_FORMAT_UNRECOGNIZED	
TDX_INCORRECT_CPUID_VALUE	Additional information is provided in RCX – R10
TDX_INCORRECT_MSR_VALUE	
TDX_INVALID_SMRR_CONFIGURATION	
TDX_INVALID_WBINVD_SCOPE	
TDX_NUM_ACTIVATED_HKIDS_NOT_SUPPORTED	
TDX_OPERAND_INVALID	
TDX_SMRR_LOCK_NOT_SUPPORTED	
TDX_SMRR_NOT_LOCKED	
TDX_SMRR_NOT_SUPPORTED	
TDX_SMRR_OVERLAPS_CMR	
TDX_SUCCESS	TDH.SYS.INIT is successful.
TDX_SYS_BUSY	The operation was invoked when another TDX module operation was in progress. The operation may be retried.
TDX_SYS_SHUTDOWN	
TDX_SYS_INIT_NOT_PENDING	

22.2.34. TDH.SYS.KEY.CONFIG Leaf

Configure the Intel TDX global private key on the current package.

Table 22.133: TDH.SYS.KEY.CONFIG Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1

Table 22.134: TDH.SYS.KEY.CONFIG Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Latency

TDH.SYS.KEY.CONFIG execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.KEY.CONFIG performs package-scope Intel TDX global private key configuration.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.135: TDH.SYS.KEY.CONFIG Operands Information

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	All Intel TDX module internal variables	N/A	RW	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. Check that TDH.SYS.CONFIG has completed successfully (PL.SYS_STATE is SYSCONFIG_DONE).

If successful, the function does the following:

2. Do the following as an atomic operation (e.g., LOCK BTS) on PL.PKG_CONFIG_BITMAP:
   2.1. Check the package has not yet been configured.
   2.2. Mark it as configured.
3. Execute PCONFIG to configure the Intel TDX global private HKID on the package with a CPU-generated random key.

PCONFIG may fail due to an entropy error or a device busy error. In these cases, the VMM should retry TDH.SYS.KEY.CONFIG.

If successful:

4. If this was the last package on which TDH.SYS.KEY.CONFIG has executed, set PL.STATE to SYS_READY.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.136: TDH.SYS.KEY.CONFIG Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_KEY_CONFIGURED	
TDX_KEY_GENERATION_FAILED	Failed to generate a random key. This is typically caused by an entropy error of the CPU's random number generator, and may be impacted by RDSEED, RDRAND or PCONFIG executing on other LPs. The operation should be retried.
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation. Specifically, key configuration may fail due to a concurrently running PCONFIG instruction.
TDX_SUCCESS	TDH.SYS.KEY.CONFIG is successful.
TDX_SYS_BUSY	The operation was invoked when another TDX module operation was in progress. The operation may be retried.
TDX_SYS_KEY_CONFIG_NOT_PENDING	
TDX_SYS_SHUTDOWN	

22.2.35. TDH.SYS.LP.INIT Leaf

Initialize the Intel TDX module at the current logical processor scope.

Table 22.137: TDH.SYS.LP.INIT Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1

Table 22.138: TDH.SYS.LP.INIT Output Operands Definition

Operand	Description		
RAX	SEAMCALL instruction return code – see 22.2.1		
RCX	Extended error information part 1 If RAX returns TDX_INCONSISTENT_CPUID_FIELD, RCX returns the applicable CPUID information as shown below. In all other cases, RCX returns 0.		
	Bits	**Name**	**Description**
	31:0	LEAF	CPUID leaf number
	63:32	SUBLEAF	CPUID sub-leaf number: if sub-leaf is not applicable, value is -1 (0xFFFFFFFF).
RDX	Extended error information part 2 If RAX returns TDX_INCONSISTENT_CPUID_FIELD, RDX returns the value masks as shown below. A bit value of 1 indicates a bit position that was checked against the same CPUID leaf value checked during TDH.SYS.INIT. In all other cases, RDX returns 0.		
	Bits	**Name**	**Description**
	31:0	MASK_EAX	Mask of the value returned by CPUID in EAX
	63:32	MASK_EBX	Mask of the value returned by CPUID in EBX
R8	Extended error information part 3 If RAX returns TDX_INCONSISTENT_CPUID_FIELD, R8 returns the value masks as shown below. A bit value of 1 indicates a bit position that was checked against the same CPUID leaf value checked during TDH.SYS.INIT. In all other cases, R8 returns 0.		
	Bits	**Name**	**Description**
	31:0	MASK_ECX	Mask of the value returned by CPUID in ECX
	63:32	MASK_EDX	Mask of the value returned by CPUID in EDX
Other	Unmodified		

Special Environment Requirements

If the IA32_TSX_CTRL MSR is supported by the CPU, as enumerated by IA32_ARCH_CAPABILITIES.TSX_CTRL (bit 7), then the values of its following bits must be 0:

- RTM_DISABLE (bit 0)
- TSX_CPUID_CLEAR (bit 1)

Leaf Function Latency

TDH.SYS.LP.INIT execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.LP.INIT performs LP-scope initialization of the Intel TDX module. This function is intended to be executed during OS/VMM boot, and thus it has relaxed latency requirements.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.139: TDH.SYS.LP.INIT Operands Information

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	All Intel TDX module internal variables	N/A	RW	Hidden	N/A	Shared	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. TDH.SYS.INIT has completed successfully (PL.SYS_STATE is SYSINIT_DONE).
2. This is the first invocation of TDH.SYS.LP.INIT on the current LP.

If successful, the function does the following:

3. Do a global EPT flush (INVEPT type 2).
4. Initialize the Intel TDX module's LP-scope variables.
5. Check the compatibility and uniformity of features and configuration. Once per LP, core or package, depending on the scope of the checked feature or configuration:
   5.1. Check features compatibility with the Intel TDX module. For example, the WBINVD scope must be the same as the scope the Intel TDX module was built to handle. In cases where the Intel TDX module supports several options, check that the features on the current LP are the same as sampled during TDH.SYS.INIT.
   5.2. Check configuration uniformity. For example, the SMRR and SMRR2 must be locked and configured in the same way as sampled during TDH.SYS.INIT.
6. Mark the current LP as initialized.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.140: TDH.SYS.LP.INIT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_BOOT_NT4_SET	
TDX_INCONSISTENT_CPUID_FIELD	Additional information is provided in RCX – R8
TDX_INCONSISTENT_MSR	
TDX_INCORRECT_MSR_VALUE	
TDX_INVALID_PKG_ID	
TDX_SEAMREPORT_NOT_AVAILABLE	

Completion Status Code	Description
TDX_SUCCESS	TDH.SYS.LP.INIT is successful.
TDX_SYS_BUSY	The operation was invoked when another TDX module operation was in progress. The operation may be retried.
TDX_SYS_LP_INIT_NOT_PENDING	
TDX_SYS_SHUTDOWN	

22.2.36. TDH.SYS.LP.SHUTDOWN Leaf

Initiate Intel TDX module shutdown, and prevent further SEAMCALLs on the current logical processor.

Table 22.141: TDH.SYS.LP.SHUTDOWN Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1

Table 22.142: TDH.SYS.LP.SHUTDOWN Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.LP.SHUTDOWN marks the Intel TDX module as being shut down (if not already in this state) and prevents further SEAMCALLs on the current LP.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.143: TDH.SYS.LP.SHUTDOWN Operands Information

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	All Intel TDX module internal variables	N/A	RW	Hidden	N/A	Shared	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. Mark the Intel TDX module as being shut down by setting PL.SYS_STATE to SYS_SHUTDOWN.
2. Prevent further SEAMCALLs on the current LP by setting the SEAM VMCS's HOST RIP field to the value of SYS_INFO_TABLE.SHUTDOWN_HOST_RIP (originally configured by the SEAMLDR).
3. Do a global EPT flush (INVEPT type 2).
   3.1. This is a defense-in-depth. In case of an Intel TDX module update, TDH.SYS.LP.INIT will do a global EPT flush.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.144: TDH.SYS.LP.SHUTDOWN Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_SUCCESS	TDH.SYS.LP.SHUTDOWN is successful.
TDX_SYS_BUSY	The operation was invoked when another TDX module operation was in progress. The operation may be retried.

22.2.37. TDH.SYS.TDMR.INIT Leaf

Partially initialize a Trust Domain Memory Region (TDMR) and its associated PAMT.

Table 22.145: TDH.SYS.TDMR.INIT Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical base address of a TDMR (HKID bits must be 0)

Table 22.146: TDH.SYS.TDMR.INIT Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
RDX	On successful completion, RDX returns the TDMR next-to-initialize address. This is the physical address of the last byte that has been initialized so far, rounded down to 1GB. In all other cases, RDX returns 0.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.SYS.TDMR.INIT partially initializes the metadata (PAMT) associated with a Trust Domain Memory Region (TDMR), while adhering to latency considerations. It can run concurrently on multiple LPs as long as each concurrent flow initializes a different TDMR. After each 1GB range of a TDMR has been initialized, that 1GB range 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.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.147: TDH.SYS.TDMR.INIT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDMR	Blob	None	None	1GB	Exclusive	N/A	N/A
Implicit	N/A	HPA	PAMT region associated with TDMR	Blob	RW	Hidden	N/A	Exclusive	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following conditions:

1. The provided TDMR start address belongs to one of the TDMRs set during TDH.SYS.INIT.
2. The TDMR has not been completely initialized yet.

If successful, the function does the following:

3. If the TDMR has been completely initialized, there is nothing to do.

Else, the function does the following:

4. Initialize the next implementation defined un-initialized number of PAMT entries. The maximum number of PAMT entries to be initialized is set to help avoid latency issues.
   4.1. PAMT_4K entries associated with a physical address that is within a reserved range are marked with PT_RSVD.
   4.2. Other PAMT_4K entries are marked with PT_NDA.
   4.3. PAMT_2M and PAMT_1G entries are marked with PT_NDA.
5. If the PAMT for a 1GB block of TDMR has been fully initialized, mark that 1GB block as ready for use. This means that 4KB pages in this 1GB block may be converted to private pages – e.g., by SEAMCALL(TDH.MEM.PAGE.ADD). This can be done concurrently with initializing other TDMRs.
6. Return the next-to-initialize address rounded down to 1GB. This is done so the host VMM will not attempt to use a 1GB block that is not fully initialized.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.148: TDH.SYS.TDMR.INIT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDH.SYS.TDMR.INIT is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TDMR_ALREADY_INITIALIZED	

22.2.38. TDH.VP.ADDCX Leaf

Add a TDVPX page to memory as a child of a given TDVPR.

Table 22.149: TDH.VP.ADDCX Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a page where the TDVPX page will be added (HKID bits must be 0)
RDX	The physical address of a TDVPR page (HKID bits must be 0)

Table 22.150: TDH.VP.ADDCX Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.ADDCX adds a TDVPX page as a child of a given TDVPR.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.151: TDH.VP.ADDCX Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPX page	Blob	RW	Opaque	4KB	Exclusive	Shared	Shared
Explicit	RDX	HPA	TDVPR page	Blob	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	HPA	TDR page	TDR	RW	Opaque	N/A	Shared	None	None
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Exclusive(i)	N/A	N/A

In addition to the explicit memory operand checks per the table above, the function checks the following conditions:

1. The TDVPR page metadata in PAMT must be correct (PT must be PT_TDVPR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. The TD has been initialized (by TDH.MNG.INIT).
5. The TD build and measurement must not have been finalized (by TDH.MR.FINALIZE).
6. The TD VCPU has not been initialized (by TDH.VP.INIT) and is not being torn down (TDVPS.STATE is VCPU_UNINITIALIZED).
7. The new TDVPX page metadata in PAMT must be correct (PT must be PT_NDA).
8. The maximum number of TDVPX pages per TD VCPUs (as enumerated by TDH.SYS.INFO) has not been exceeded.

If successful, the function does the following:

9. Zero out the TDVPX page contents using direct writes (MOVDIR64B).
10. Increment the VCPU's TDVPX counter, and set a pointer in the parent TDVPR page to the new TDVPX page.

11. Increment TDR.CHLDCNT.
12. Initialize the TDVPX page metadata in PAMT.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.152: TDH.VP.ADDCX Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.ADDCX is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_FINALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TDVPX_NUM_INCORRECT	
TDX_VCPU_STATE_INCORRECT	

22.2.39. TDH.VP.CREATE Leaf

Create a guest TD VCPU and its root TDVPR page.

Table 22.153: TDH.VP.CREATE Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a page where TDVPR will be added (HKID bits must be 0)
RDX	The physical address of the owner TDR page (HKID bits must be 0)

Table 22.154: TDH.VP.CREATE Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.CREATE begins the build of a new guest TD VCPU. It adds a TDVPR page as a child of a TDR page.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.155: TDH.VP.CREATE Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPR page	Blob	RW	Opaque	4KB	Exclusive	Shared	Shared
Explicit	RDX	HPA	TDR page	TDR	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	4KB	Shared(i)	N/A	N/A

In addition to the explicit memory operand checks per the table above, the function checks the following conditions:

1. The TDR page metadata in PAMT must be correct (PT must be PT_TDR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD build and measurement must not have been finalized (by TDH.MR.FINALIZE).
6. The TDVPR page metadata in PAMT must be correct (PT must be PT_NDA).

If successful, the function does the following:

7. Zero out the TDVPR page contents using direct write (MOVDIR64B).
8. Increment TDR.CHLDCNT.
9. Initialize the TDVPR management fields.
10. Initialize the TDVPR page metadata in PAMT.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.156: TDH.VP.CREATE Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.CREATE is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_FINALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	

22.2.40. TDH.VP.ENTER Leaf

Enter TDX non-root operation.

From the host VMM's point of view, TDH.VP.ENTER is a complex operation that normally involves TD entry followed by a TD exit. Therefore, input and output operands are specified by multiple tables below.

The following table details TDH.VP.ENTER input operands for **initial entry** or following a **previous asynchronous TD exit**.

Table 22.157: : TDH.VP.ENTER Input Operands Definition for Initial Entry or Following a Previous Asynchronous TD Exit

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of the TD VCPU's TDVPR page (HKID bits must be 0)

The following table details TDH.VP.ENTER input operands for following a **previous synchronous TD exit**.

Table 22.158: TDH.VP.ENTER Input Operands Definition Following a Previous TDCALL(TDG.VP.VMCALL)

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of the TD VCPU's TDVPR page
RBX, RDX, RBP, RDI, RSI, R8 – R15	If the corresponding bit of RCX at the previous TD exit (i.e., previous TDH.VP.ENTER termination) was set to 1, the register value is passed as-is to the guest TD – see the description of TDG.VP.VMCALL in 22.3.10 for details. Else, the register value is not used as an input.
XMM0 – XMM15	If the corresponding bit of RCX at the previous TD exit (i.e., previous TDH.VP.ENTER termination) was set to 1, the register value is passed as-is to the guest TD – see the description of TDG.VP.VMCALL in 22.3.10 for details. Else, the register value is not used as an input.

The following table details TDH.VP.ENTER output operands when an error occurs, and the interface function returns **without entering the TD**.

Table 22.159: TDH.VP.ENTER Output Operands Definition on Error (No TD Entry)

Operand	Description
RAX	SEAMCALL instruction return code – see 17.3.2
Other GPRs	Unmodified
Extended State	Any extended state that the TD is allowed to use (per TDCS.XFAM) may be cleared to its architectural INIT state.

The following table details TDH.VP.ENTER output operands on when TD entry succeeds, and later an **asynchronous TD exit** occurs due to a **VMX architectural exit reason**.

Table 22.160: TDH.VP.ENTER Output Operands Definition on Asynchronous TD Exits Following a TD Entry

Operand	TD Exit Information	Description
RAX	Status and Exit Reason	SEAMCALL instruction return code – see 17.3.2 Note the special values of the DETAILS_L1 field in bits 39:32 which indicate that the VCPU or the whole TD is corrupt and cannot be re-entered: • TDX_NON_RECOVERABLE_VCPU • TDX_NON_RECOVERABLE_TD The DETAILS_L2 field in bits 31:0 contain the VMCS exit reason.
RCX	Exit Qualification	Format is similar to the VMCS exit qualification. When exit is due to EPT violation, bits 12-7 of the exit qualification are cleared to 0.
RDX	Extended Exit Qualification	Additional non-VMX, TDX-specific information – see 20.6.1
R8	Guest Physical Address	When exit is due to EPT violation or EPT misconfiguration, format is similar to the VMCS guest-physical address, except that bits 11:0 are cleared to 0. In other cases, R8 is cleared to 0.
R9	VM-Exit Interruption Information	When exit is due to a vectored event, format of bits 31:0 is similar to the VMCS VM-exit interruption information. Bits 63:32 are cleared to 0. In other cases, R9 is cleared to 0.
RBX, RSI, RDI, R10 – R15	None	Cleared to 0
Extended State	Any extended state that the TD is allowed to use (per TDCS.XFAM) is cleared to its architectural INIT state.	

The following table details TDH.VP.ENTER output operands on when TD entry succeeds, and later a **synchronous TD exit**, triggered by **TDG.VP.VMCALL**, occurs.

Table 22.161: TDH.VP.ENTER Output Operands Definition on TDCALL(TDG.VP.VMCALL) Following a TD Entry

Operand	Description
RAX	SEAMCALL instruction return code – see 17.3.2 The DETAILS_L2 field in bits 31:0 contain the VMCS exit reason, indicating TDCALL (77).
RCX	Value as passed in to TDCALL(TDG.VP.VMCALL) by the guest TD: indicates which part of the guest TD GPR and XMM state is passed as-is to the VMM and back. For details, see the description of TDG.VP.VMCALL in 22.3.10.
RBX, RDX, RBP, RDI, RSI, R8 – R15	If the corresponding bit in RCX is set to 1, the register value is passed as-is from the guest TD's input to TDG.VP.VMCALL. Else, the register value cleared to 0.
XMM0 – XMM15	If the corresponding bit in RCX is set to 1, the register value is passed as-is from the guest TD's input to TDG.VP.VMCALL. Else, the register value cleared to 0.
Extended State except XMM	Any extended state, except XMM, that the TD is allowed to use (per TDCS.XFAM) is cleared to its architectural INIT state.

CPU State Preservation Following a Successful TD Entry and a TD Exit

Following a successful TD entry and a TD exit, some CPU state is modified:

- Registers CR2, DR0, DR1, DR2, DR3, DR6 and DR7 are set to their architectural INIT value.
- XCR0 is set to the TD's user-mode feature bits of XFAM (bits 7:0, 9).
- Multiple MSRs are set as described in Table 22.162 below. In this table, Init(condition) means that the MSR is set to its INIT value if the condition is true, else the MSR is unmodified.

Table 22.162: MSRs that may be Modified by TDH.VP.ENTER

MSR Index Range (Hex)				
First (H)	**Last (H)**	**Size (H)**	**MSR Architectural Name**	**MSR Preservation across TDH.VP.ENTER**
0x00E1	0x00E1	0x1	IA32_UMWAIT_CONTROL	Init(virt. CPUID(7,0).ECX[5])
0x0186	0x018D	0x8	IA32_PERFEVTSELx	Init(PERFMON)
0x01A6	0x01A7	0x2	MSR_OFFCORE_RSPx	Init(PERFMON)
0x01C4	0x01C4	0x1	IA32_XFD	Init(virt. CPUID(0xD,0x1).EAX[4])
0x01C5	0x01C5	0x1	IA32_XFD_ERR	Init(virt. CPUID(0xD,0x1).EAX[4])
0x01D9	0x01D9	0x1	IA32_DEBUGCTL	INIT, except for the following bits which are preserved: Bit 1 (BTF) Bit 12 (FREEZE_PERFMON_ON_PMI) Bit 14 (FREEZE_WHILE_SMM)
0x0309	0x030C	0x4	IA32_FIXED_CTRx	Init(PERFMON)
0x0329	0x0329	0x1	IA32_PERF_METRICS	Init(PERFMON)
0x038D	0x038D	0x1	IA32_FIXED_CTR_CTRL	Init(PERFMON)
0x038E	0x038E	0x1	IA32_PERF_GLOBAL_STATUS	Init(PERFMON)
0x038F	0x038F	0x1	IA32_PERF_GLOBAL_CTRL	Init(PERFMON)
0x03F1	0x03F1	0x1	IA32_PEBS_ENABLE	Init(PERFMON)
0x03F2	0x03F2	0x1	MSR_PEBS_DATA_CFG	Init(PERFMON)
0x03F6	0x03F6	0x1	MSR_PEBS_LD_LAT	Init(PERFMON)
0x03F7	0x03F7	0x1	MSR_PEBS_FRONTEND	Init(PERFMON)
0x04C1	0x04C8	0x8	IA32_A_PMCx	Init(PERFMON)
0x0560	0x0560	0x1	IA32_RTIT_OUTPUT_BASE	Init(XFAM(8))
0x0561	0x0561	0x1	IA32_RTIT_OUTPUT_MASK_PTRS	Init(XFAM(8))
0x0570	0x0570	0x1	IA32_RTIT_CTL	Init(XFAM(8))
0x0571	0x0571	0x1	IA32_RTIT_STATUS	Init(XFAM(8))
0x0572	0x0572	0x1	IA32_RTIT_CR3_MATCH	Init(XFAM(8))
0x0580	0x0580	0x1	IA32_RTIT_ADDR0_A	Init(XFAM(8))
0x0581	0x0581	0x1	IA32_RTIT_ADDR0_B	Init(XFAM(8))
0x0582	0x0582	0x1	IA32_RTIT_ADDR1_A	Init(XFAM(8))
0x0583	0x0583	0x1	IA32_RTIT_ADDR1_B	Init(XFAM(8))
0x0584	0x0584	0x1	IA32_RTIT_ADDR2_A	Init(XFAM(8))
0x0585	0x0585	0x1	IA32_RTIT_ADDR2_B	Init(XFAM(8))
0x0586	0x0586	0x1	IA32_RTIT_ADDR3_A	Init(XFAM(8))
0x0587	0x0587	0x1	IA32_RTIT_ADDR3_B	Init(XFAM(8))
0x0600	0x0600	0x1	IA32_DS_AREA	INIT
0x06A0	0x06A0	0x1	IA32_U_CET	Init(XFAM[11] | XFAM[12])
0x06A4	0x06A4	0x1	IA32_PL0_SSP	Init(XFAM[11] | XFAM[12])
0x06A5	0x06A5	0x1	IA32_PL1_SSP	Init(XFAM[11] | XFAM[12])
0x06A6	0x06A6	0x1	IA32_PL2_SSP	Init(XFAM[11] | XFAM[12])
0x06A7	0x06A7	0x1	IA32_PL3_SSP	Init(XFAM[11] | XFAM[12])
0x0985	0x0985	0x1	IA32_UINT_RR	Init(XFAM[14])
0x0986	0x0986	0x1	IA32_UINT_HANDLER	Init(XFAM[14])
0x0987	0x0987	0x1	IA32_UINT_STACKADJUST	Init(XFAM[14])
0x0988	0x0988	0x1	IA32_UINT_MISC	Init(XFAM[14])
0x0989	0x0989	0x1	IA32_UINT_PD	Init(XFAM[14])
0x098A	0x098A	0x1	IA32_UINT_TT	Init(XFAM[14])

MSR Index Range (Hex)				
First (H)	**Last (H)**	**Size (H)**	**MSR Architectural Name**	**MSR Preservation across TDH.VP.ENTER**
0x0DA0	0x0DA0	0x1	IA32_XSS	Supervisor-mode feature bits of XFAM (bits 8, 16:10)
0x1200	0x12FF	0x100	IA32_LBR_INFO	Init(XFAM[15])
0x14CE	0x14CE	0x1	IA32_LBR_CTL	Init(XFAM[15])
0x14CF	0x14CF	0x1	IA32_LBR_DEPTH	Init(XFAM[15])
0x1500	0x15FF	0x100	IA32_LBR_FROM_IP	Init(XFAM[15])
0x1600	0x16FF	0x100	IA32_LBR_TO_IP	Init(XFAM[15])
0xC0000081	0xC0000081	0x1	IA32_STAR	INIT
0xC0000082	0xC0000082	0x1	IA32_LSTAR	INIT
0xC0000084	0xC0000084	0x1	IA32_FMASK	INIT
0xC0000102	0xC0000102	0x1	IA32_KERNEL_GS_BASE	INIT
0xC0000103	0xC0000103	0x1	IA32_TSC_AUX	INIT

Special Environment Requirements

The value read from IA32_TSC_ADJUST MSR must be the same as it was during TDH.SYS.INIT.

If the IA32_TSX_CTRL MSR is supported by the CPU, as enumerated by IA32_ARCH_CAPABILITIES.TSX_CTRL (bit 7), then the values of its following bits must be 0:

- RTM_DISABLE (bit 0)
- TSX_CPUID_CLEAR (bit 1)

Leaf Function Latency

In some cases (e.g., suspected single/zero step attack mitigation), TDH.VP.ENTER execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.ENTER enters TDX non-root operation.

VCPU Association: TDH.VP.ENTER associates the target TD VCPU with the current LP. This requires that the VCPU will not be associated with another LP. For details, see 9.3.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.163: TDH.VP.ENTER Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPR page	TDVPS	RW	Opaque	4KB	Shared[13]	Shared[13]	Shared[13]
Implicit	N/A	HPA	TDR page	TDR	RW	Opaque	N/A	Shared[13]	N/A	N/A
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)[13]	N/A	N/A
Implicit	N/A	N/A	TDCS TLB Tracking Fields	N/A	RW	Opaque	N/A	Shared[14]	N/A	N/A

[13] The shared locking of TDVPS, TDR and TDCS (but not the TDCS epoch tracking fields) is for the whole duration of running in TDX non-root mode; the locks are released on TD exit.

[14] The locking of SEPT tree and the TLB tracking fields is until before entering TDX non-root mode; the locks are released before VM entry into the TD VCPU.

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Implicit	N/A	N/A	SEPT tree	N/A	R	Opaque	N/A	Exclusive[14]	N/A	N/A

In addition to the explicit memory operand checks per the table above, the function checks the following conditions:

1. The TDVPR page metadata in PAMT must be correct (PT must be PT_TDVPR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The TD build and measurement must have been finalized (by TDH.MR.FINALIZE).

If successful, the function does the following:

6. Associate the VCPU with the current LP and update TD VMCS.
    - 6.1. Check that the VCPU has been initialized and is not being torn down.
    - 6.2. Atomically check that the VCPU is not associated with another LP, and associate it with the current LP.
    - 6.3. If the TD's ephemeral HKID has changed since last VM entry, update all TD VMCS physical pointers and the TD HKID execution control.
    - 6.4. Update the TD VMCS host state fields with any Intel TDX module LP-specific values.

If passed:

7. If the TD VCPU to be entered is different than the last TD VCPU entered on the current LP, issue an indirect branch prediction barrier command to the CPU by writing to the IA32_PRED_CMD MSR with the IBPB bit set.
8. Update the TLB tracking state. This is done as a critical section allowing concurrent TDH.VP.ENTERs but no concurrent TDH.MEM.TRACK. A concurrent TDH.MEM.TRACK may cause this locking to fail; in this case, the caller is expected to retry TDH.VP.ENTER.
    - 8.1. Lock the TDCS epoch tracking fields in shared mode.
    - 8.2. Sample the TD's epoch counter (TDCS.TD_EPOCH) into the VCPU's TDVPS.VCPU_EPOCH.
    - 8.3. Atomically increment the TD's REFCOUNT that is associated with the sampled epoch (TDCS.REFCOUNT[TD_EPOCH % 2]).
    - 8.4. Release the shared mode locking of the epoch tracking fields.

If successful:

9. Set TDVPS.VCPU_STATE to VCPU_ACTIVE.
10. Restore guest TD state:
    - 10.1. If previous TD exit was due to a TDG.VP.VMCALL:
        - 10.1.1. Restore guest XMM and GPR state that is not passed as-is from the host VMM, as controlled by the value of guest TD RCX input to TDG.VP.VMCALL.
        - 10.1.2. Set guest RAX to 0.
    - 10.2. Else (TD exit was an asynchronous exit):
        - 10.2.1. Restore CPU extended state from TDVPS (per TDCS.XFAM).
    - 10.3. Restore other guest state from TDVPS.
11. Execute VMLAUNCH or VMRESUME depending on whether this VCPU has been launched on this LP since its last association with the LP (TVPS.VMLAUNCH).

Note: Logically, from the point of view of the host VMM, a successful TDH.VP.ENTER is terminated by the next TD exit.

Completion Status Codes

In case of successful execution (which resulted in the TD guest running and then exiting), the status code value in RAX is encoded the same as the VMX Exit reason – see 17.3.2 for details.

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.164: TDH.VP.ENTER Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_BOOT_NT4_SET	
TDX_INCONSISTENT_MSR	Specifically, IA32_TSC_ADJUST MSR value must be the same as sampled by TDH.SYS.INIT.
TDX_NON_RECOVERABLE_TD	TDH.VP.ENTER launched or resumed TD VCPU operation (TDX non-root mode) – followed later by a TD exit. The TD state is non-recoverable – further TD entry is prohibited. Exit reason is in RAX bits 31:0.
TDX_NON_RECOVERABLE_TD_FATAL	TDH.VP.ENTER launched or resumed TD VCPU operation (TDX non-root mode) – followed later by a TD exit. The TD state is non-recoverable – further TD entry is prohibited, and TD private memory can't be accessed. Exit reason is in RAX bits 31:0.
TDX_NON_RECOVERABLE_TD_WRONG_APIC_MODE	The host VMM is running with local APIC mode is not supported for TD operation – further TD entry is prohibited. TDH.VP.ENTER launched or resumed TD VCPU operation (TDX non-root mode) – followed later by a TD exit. The. Exit reason is in RAX bits 31:0.
TDX_NON_RECOVERABLE_VCPU	TDH.VP.ENTER launched or resumed TD VCPU operation (TDX non-root mode) – followed later by a TD exit. The TD VCPU state is non-recoverable – further TD entry to this VCPU is prohibited. Exit reason is in RAX bits 31:0.
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation. Note the special case where the indicated operand is TLB_EPOCH. This may happen due to a conflict with TDH.MEM.TRACK. The host VMM should retry TDH.VP.ENTER. Another special case is where the indicated operand is SEPT. In some cases, TDH.VP.ENTER may acquire exclusive access on the SEPT tree for a short period of time, and may fail due to a concurrent operation . The host VMM should retry TDH.VP.ENTER.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.ENTER launched or resumed TD VCPU operation (TDX non-root mode) – followed later by a TD exit. Exit reason is in RAX bits 31:0.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_FINALIZED	

Completion Status Code	Description
TDX_TD_NOT_INITIALIZED	
TDX_TDVPX_NUM_INCORRECT	
TDX_TSC_ROLLBACK	
TDX_VCPU_ASSOCIATED	
TDX_VCPU_STATE_INCORRECT	

22.2.41. TDH.VP.FLUSH Leaf

Flush the address translation caches and cached TD VMCS associated with a TD VCPU on the current logical processor.

Table 22.165: TDH.VP.FLUSH Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDVPR page (HKID bits must be 0)

Table 22.166: TDH.VP.FLUSH Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.FLUSH flushes the address translation caches and cached TD VMCS associated with a TD VCPU on the current LP. It then marks the VCPU as not associated with any LP.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.167: TDH.VP.FLUSH Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPR page	TDVPS	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	HPA	TDR page	TDR	R	Opaque	N/A	Shared	N/A	N/A
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDVPR page metadata in PAMT must be correct (PT must be PT_TDVPR).
2. TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED.
3. The current VCPU must be currently associated with the current LP.

If the above checks pass, the function does the following:

4. Flush the TLB context and extended paging structure (EPxE) caches associated with the TD using INVEPT single-context invalidation (type 1).
5. Flush the cached TD VMCS content to TDVPS using VMCLEAR.
6. Mark the current VCPU as not associated with any LP.
7. Atomically decrement (using LOCK XADD) the associated VCPUs counter (TDCS.NUM_ASSOC_VCPUS).

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.168: TDH.VP.FLUSH Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.FLUSH is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TDVPX_NUM_INCORRECT	
TDX_VCPU_NOT_ASSOCIATED	

22.2.42. TDH.VP.INIT Leaf

Initialize the saved state of a TD VCPU.

Operands

Table 22.169: TDH.VP.INIT Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDVPR page (HKID bits must be 0)
RDX	Initial value of the guest TD VCPU RCX

Table 22.170: TDH.VP.INIT Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
Other	Unmodified

Leaf Function Latency

TDH.VP.INIT execution time may be longer than most TDX module interface functions execution time. No interrupts (including NMI and SMI) are processed by the logical processor during that time.

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.INIT initialized the saved state of a VCPU in the TDVPR and TDPX pages.

VCPU Association: TDH.VP.INIT associates the target TD VCPU with the current LP – for details, see 9.3.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.171: TDH.VP.INIT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPR page	TDVPS	RW	Opaque	4KB	Exclusive	Shared	Shared
Implicit	N/A	HPA	TDR page	TDR	R	Opaque	4KB	Shared	N/A	N/A
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	4KB	Shared(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDVPR page metadata in PAMT must be correct (PT must be PT_TDVPR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. The TD build and measurement must not have been finalized (by TDH.MR.FINALIZE).
5. The TD VCPU has not been initialized (by TDH.VP.INIT) and is not being torn down (TDVPS.STATE is VCPU_UNINITIALIZED).
6. The number of TDVPX pages associated with the TDVPR page is correct.

7. Atomically Increment the TD's VCPU counter (TDCS.NUM_VCPUS), and check that maximum number of VCPUs (TDCS.MAX_VCPUS) has not been exceeded.

If successful, the function does the following:

8. Assign a unique sequential identifier to the VCPU.
9. Initialize the VCPU state fields in the logical TDVPS structure (TDVPR and TDVPX pages).
10. Set the VCPU state VCPU_READY_ASYNC since the first TD entry is the same as TD entry following an asynchronous TD exit.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.172: TDH.VP.INIT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_INCONSISTENT_MSR	Specifically, IA32_TSC_ADJUST MSR value must be the same as sampled by TDH.SYS.INIT.
TDX_MAX_VCPUS_EXCEEDED	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.INIT is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_FINALIZED	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TDVPX_NUM_INCORRECT	
TDX_VCPU_STATE_INCORRECT	

22.2.43. TDH.VP.RD Leaf

Read a TDVPS field.

Table 22.173: TDH.VP.RD Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDVPR page (HKID bits must be 0)
RDX	Field code – see 20.9.3

Table 22.174: TDH.VP.RD Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
R8	Field content In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.RD reads a TDVPS field, given its field code. Reading is subject to the field's readability (per the TD's ATTRIBUTES.DEBUG bit).

VCPU Association: TDH.VP.RD associates the target TD VCPU with the current LP. This requires that the VCPU will not be associated with another LP – for details, see 9.3.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.175: TDH.VP.RD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPR page	TDVPS	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	HPA	TDR page	TDR	R	Opaque	N/A	Shared	N/A	N/A
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDVPR page metadata in PAMT must be correct (PT must be PT_TDVPR).
2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. The TD has been initialized (by TDH.MNG.INIT).
5. The provided field code is valid.
6. The provided TDVPS field is readable per the TD's debug attribute (TDCS.ATTRIBUTES.DEBUG).

If successful, the function does the following:

7. Associate the VCPU with the current LP, and update TD VMCS.
   7.1. Check that the VCPU has been initialized and is not being torn down.
   7.2. Atomically check that the VCPU is not associated with another LP, and associate it with the current LP.
   7.3. If the TD's ephemeral HKID has changed since last VM entry, update all TD VMCS physical pointers and the TD HKID execution control.
   7.4. Update the TD VMCS host state fields with any Intel TDX module LP-specific values.

If passed:

8. Derive the field attributes (read mask, VMCS field code or offset in TDVPS) from the field code provided in RDX per the TD's ATTRIBUTE.DEBUG. If the read mask is 0, then fail; the field in not readable.

If passed:

9. Read the field value (VMREAD from TD VMCS, directly from other TDVPS areas).
10. Mask out the field value with the read mask derived earlier and return in R8.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.176: TDH.VP.RD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FIELD_NOT_READABLE	
TDX_OPERAND_ADDR_RANGE_ERROR	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.RD is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TDVPX_NUM_INCORRECT	
TDX_VCPU_ASSOCIATED	
TDX_VCPU_STATE_INCORRECT	

22.2.44. TDH.VP.WR Leaf

Write a TDVPS field.

Table 22.177: TDH.VP.WR Input Operands Definition

Operand	Description
RAX	SEAMCALL instruction leaf number – see 22.2.1
RCX	The physical address of a TDVPR page (HKID bits must be 0)
RDX	Field code – see 20.9.3
R8	64b value to write to the field
R9	64b mask to indicate which bits of the value in R8 are to be written to the field

Table 22.178: TDH.VP.WR Output Operands Definition

Operand	Description
RAX	SEAMCALL instruction return code – see 22.2.1
R8	Previous content of the field In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDH.VP.WR writes a TDVPS field, given its field code. The specific bits of the value (R8) are written as specified by the write mask (R9). Writing is subject to the field's writability (per the TD's ATTRIBUTES.DEBUG bit). Writing of specific fields is also subject to additional rules as detailed in 21.2.

TDH.VP.WR returns the previous content of the field masked by the field's readability (per the TD's ATTRIBUTES.DEBUG bit).

VCPU Association: TDH.VP.WR associates the target TD VCPU with the current LP. This requires that the VCPU will not be associated with another LP – for details, see 9.3.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.179: TDH.VP.WR Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions		
								Operand	Contain. 2MB	Contain. 1GB
Explicit	RCX	HPA	TDVPR page	TDVPS	RW	Opaque	4KB	Shared	Shared	Shared
Implicit	N/A	HPA	TDR page	TDR	R	Opaque	N/A	Shared	N/A	N/A
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)	N/A	N/A

In addition to the memory operand checks per the table above, the function checks the following:

1. The TDVPR page metadata in PAMT must be correct (PT must be PT_TDVPR).

2. The TD is not in a FATAL state (TDR.FATAL is FALSE).
3. The TD keys are configured on the hardware (TDR.LIFECYCLE_STATE is TD_KEYS_CONFIGURED).
4. TDCS must have been initialized (TDR.INIT is TRUE).
5. The provided field code is valid.
6. The provided TDVPS field is writable per the TD's debug attribute (TDCS.ATTRIBUTES.DEBUG).

If successful, the function does the following:

7. Associate the VCPU with the current LP and update TD VMCS.
   7.1. Check that the VCPU has been initialized and is not being torn down.
   7.2. Atomically check that the VCPU is not associated with another LP, and associate it with the current LP.
   7.3. If the TD's ephemeral HKID has changed since last VM entry, update all TD VMCS physical pointers and the TD HKID execution control.
   7.4. Update the TD VMCS host state fields with any Intel TDX module LP-specific values.

If passed:

8. Derive the field attributes (read mask, write mask, VMCS field code or offset in TDVPS) from the field code provided in RDX per the TD's ATTRIBUTE.DEBUG.

If passed:

9. Derive the effective write mask by bitwise-anding the write mask derived above with the write mask provided in R9. If the effective write mask is 0, then fail; the field in not writable.

If passed:

10. Read the field value. For a TD VMCS field, use VMREAD. For other fields, read directly from the TDVPS.
11. Update the field value from the input value in R8, per the effective write mask, and write the field (use VMWRITE to write TD VMCS, write directly to other TDVPS areas).
    11.1. Writes of some fields are subject to rules, as detailed per field in 21.2 – e.g., the value of fields that contain Shared physical address, such as the Shared EPT Pointer, must have a Shared HKID value and must comply with some alignment rules.
    11.2. In most cases, writes of guest state fields are subject to the same rules as if the write is done by the guest itself – e.g., writing to guest CR4 is subject to the rules described in 10.6.2. If the write operation is illegal, TDH.VP.WR fails and returns a proper error code.
    11.3. In debug mode (ATTRIBUTES.DEBUG == 1), there are some TDVPS fields where the TDH.VP.WR does not check whether the written values are architecturally valid. It is the responsibility of the host VMM, and failing to do so will later cause a VM entry failure leading to a fatal shutdown of the Intel TDX module. The security of any guest TD is not impacted.
    11.4. In other cases, in debug mode (ATTRIBUTES.DEBUG == 1), TDH.VP.WR allows setting of TDVPS fields to values that may impact the correct operation of the TD under debug. It is the responsibility of the host VMM to take this into consideration.
    - TDH.VP.WR is allowed to enable BTM by setting guest IA32_DEBUGCTL[7:6] to 0x1 (see 14.1.3).
    - TDH.VP.WR is allowed to modify the state of IA32_DEBUCTL[13] (ENABLE_UNCORE_PMI) (see 14.1.2.2).
    - TDH.VP.WR is allowed to enable VM exits on exceptions other than MCE by setting the TD VMCS exception bitmap execution control. The Intel TDX module does not take this into account when handling VM exits that occur during event delivery.

If passed:

12. Mask out the previous field value with the read mask derived earlier, and return in R8.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.180: TDH.VP.WR Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FIELD_NOT_WRITABLE	
TDX_OPERAND_ADDR_RANGE_ERROR	

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_OPERAND_PAGE_METADATA_INCORRECT	
TDX_SUCCESS	TDH.VP.WR is successful.
TDX_SYS_NOT_READY	
TDX_SYS_SHUTDOWN	
TDX_TD_FATAL	
TDX_TD_KEYS_NOT_CONFIGURED	
TDX_TD_NOT_INITIALIZED	
TDX_TD_VMCS_FIELD_NOT_INITIALIZED	
TDX_TDVPX_NUM_INCORRECT	
TDX_VCPU_ASSOCIATED	
TDX_VCPU_STATE_INCORRECT	

22.3. Guest-Side (TDCALL) Interface Functions

The TDCALL instruction causes a VM exit to the Intel TDX module. It is used to call guest-side Intel TDX functions, either local or a TD exit to the host VMM, as selected by RAX.

22.3.1. TDCALL Instruction (Common)

This section describes the common functionality of TDCALL. Leaf functions are described in the following sections. As used by the Intel TDX module, TDCALL is allowed only in 64b mode.

Table 22.181: TDCALL Input Operands Definition

Operand	Description
RAX	Leaf number – see Table 22.183 below.
Other	See individual TDCALL leaf functions.

Table 22.182: TDCALL Output Operands Definition

Operand	Description
RAX	Instruction return code, indicating the outcome of execution of the instruction – see 17.3.2 for details.
Other	See individual TDCALL leaf functions.

Table 22.183: TDCALL Instruction Leaf Numbers Definition

Leaf Number	Interface Function Name
0	TDG.VP.VMCALL
1	TDG.VP.INFO
2	TDG.MR.RTMR.EXTEND
3	TDG.VP.VEINFO.GET
4	TDG.MR.REPORT
5	TDG.VP.CPUIDVE.SET
6	TDG.MEM.PAGE.ACCEPT
7	TDG.VM.RD
8	TDG.VM.WR

Instruction Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

This section describes how TDCALL leaf functions are implemented by the Intel TDX module.

On VM exit, the Intel TDX module performs the following checks:

1. If the CPU mode is not 64b ((IA32_EFER.LMA == 1) && (CS.L == 1)), the Intel TDX module injects a #GP(0) fault into the guest TD.
2. If the leaf number in RAX is not supported by the Intel TDX module, it returns a TDX_OPERAND_INVALID(0) status code in RAX.

If all checks pass, the Intel TDX module calls the leaf function according to the leaf number in RAX – see the following sections for individual leaf function details.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.184: TDCALL Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_SUCCESS	TDCALL is successful.
TDX_OPERAND_INVALID	Illegal leaf number
Other	See individual leaf functions

22.3.2. TDG.MEM.PAGE.ACCEPT Leaf

Accept a pending private page, and initialize the page to all-0 using the TD ephemeral private key.

Table 22.185: TDG.MEM.PAGE.ACCEPT Input Operands Definition

Operand	Description		
RAX	TDCALL instruction leaf number – see 22.3.1		
RCX	EPT mapping information:		
	Bits	**Name**	**Description**
	2:0	Level	Level of the Secure EPT entry that maps the private page to be accepted: either 0 (4KB) or 1 (2MB) – see 20.5.1.
	11:3	Reserved	Reserved: must be 0
	51:12	GPA	Bits 51:12 of the guest physical address of the private page to be removed
	63:52	Reserved	Reserved: must be 0

Table 22.186: TDG.MEM.PAGE.ACCEPT Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code – see 22.3.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

Accept a pending private page, previously added by TDH.MEM.PAGE.AUG, into the TD. Initialize the page to 0.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.187 TDG.MEM.PAGE.ACCEPT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Explicit	RCX	GPA	TD private page	Blob	RW	Private	$2^{12+9*Level}$ Bytes	None
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDVPS structure	TDVPS	RW	Opaque	N/A	Shared
Implicit	N/A	N/A	Secure EPT tree	N/A	RW	Private	N/A	None
Implicit	N/A	GPA	Secure EPT entry	SEPT Entry	RW	Private	N/A	Exclusive[15], Transaction

[15] Guest-side only

TDG.MEM.PAGE.ACCEPT checks the memory operands per the table above when applicable during its flow. The text below does not explicitly mention those checks, except when necessary.

In addition to the memory operand checks per the table above, the function does the following (no specific order is implied):

1. Walk the Secure EPT based on the GPA operand and requested level. The walk is successful if arrived at a leaf entry whose state is SEPT_PENDING. In case of error, return a status code or TD exit as described in Table 8.3.

If successful, do the following:

2. Loop until the whole page has been initialized, or until interrupted:
   2.1. Initialize the next 4KB chunk to 0 using the TD's ephemeral private HKID and direct writes (MOVDIR64B).
   2.2. If not done and there is a pending interrupt, abort TDG.MEM.PAGE.ACCEPT and resume the guest TD without updating RIP and any GPR.

If done initializing the page, do the following:

3. Atomically (using LOCK CMPXCHG), check that the entry state is still SEPT_PENDING, and set it to SEPT_PRESENT.
   3.1. If failed (a concurrent host-side function may have changed the Secure EPT entry state), do a TD exit with an EPT Violation exit reason and a NOT_PENDING indication in the extended exit qualification.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.188: TDG.MEM.PAGE.ACCEPT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_INVALID	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation. Specifically, it may indicate that a concurrent TDG.MEM.PAGE.ACCEPT is using the same Secure EPT entry
TDX_PAGE_ALREADY_ACCEPTED	
TDX_PAGE_SIZE_MISMATCH	Requested page size is 2MB, but the page GPA is not mapped at 2MB size
TDX_SUCCESS	TDG.MEM.PAGE.ACCEPT is successful.

22.3.3. TDG.MR.REPORT Leaf

TDG.MR.REPORT creates a TDREPORT_STRUCT structure that contains the measurements/configuration information of the guest TD that called the function, measurements/configuration information of the Intel TDX module and a REPORTMACSTRUCT.

Table 22.189: TDG.MR.REPORT Input Operands Definition

Operand	Description		
RAX	TDCALL instruction leaf number – see 22.3.1		
RCX	1024B-aligned guest physical address of newly created report structure		
RDX	64B-aligned guest physical address of additional data to be signed		
R8	**Bits**	**Name**	**Description**
	7:0	Report sub type	Must be 0
	63:8	Reserved	Reserved: must be 0

Table 22.190: TDG.MR.REPORT Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code – see 22.3.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

This function creates a TDREPORT_STRUCT structure that contains the measurements/configuration information of the guest TD that called the function, measurements/configuration information of the Intel TDX module and a REPORTMACSTRUCT. The REPORTMACSTRUCT is integrity-protected with a MAC, and it contains the hash of the measurements and configuration as well as additional REPORTDATA provided by the TD software.

Additional REPORTDATA, a 64-byte value, is provided by the guest TD to be included in the TDG.MR.REPORT.

Note: Although not enforced by TDG.MR.REPORT, the guest TD should normally place REPORTDATA in private memory to help ensure secure report generation.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.191: TDG.MR.REPORT Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Explicit	RCX	GPA	Output report	TDREPORT_STRUCT	RW	Private/ Shared	1024B	None
Explicit	RDX	GPA	Input report data	REPORTDATA	R	Private/ Shared	64B	None
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDCS.RTMR	SHA384_HASH	N/A	Opaque	N/A	Shared

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDVPS structure	TDVPS	None	Opaque	N/A	Shared

In addition to the memory operand checks per the table above, the function checks the following conditions (no specific order is implied):

1. R8 must specify report sub type 0.

If passed, the function does the following:

2. Assemble a report type structure based on the report sub type provided in R8.
3. Assemble the output report's TDG.VP.INFO fields from the TDCS reported fields (ATTRIBUTES, XFAM, MRTD, MRCONFIGID, MROWNER, MROWNERCONFIG and RTMRs).
4. Calculate a SHA384 hash over TDG.VP.INFO.
5. Execute SEAMREPORT to complete the output report, based on the input report data, the TDG.VP.INFO hash calculated above and the report type structure.
6. Write the output report to memory.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.192: TDG.MR.REPORT Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDG.MR.REPORT is successful.

22.3.4. TDG.MR.RTMR.EXTEND Leaf

Extend a TDCS.RTMR measurement register.

Table 22.193: TDG.MR.RTMR.EXTEND Input Operands Definition

Operand	Description
RAX	TDCALL instruction leaf number – see 22.3.1
RCX	64B-aligned guest physical address of a 48B extension data buffer
RDX	Index of the measurement register to be extended

Table 22.194: TDG.MR.RTMR.EXTEND Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code – see 22.3.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

This function extends one of the RTMR measurement registers in TDCS with the provided extension data in memory.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.195 TDG.MR.RTMR.EXTEND Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Explicit	RCX	GPA	EXTEND_DATA	Blob	R	Private	64B	None
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	RW	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDCS.RTMR	SHA384_HASH	N/A	Opaque	N/A	Exclusive
Implicit	N/A	N/A	TDVPR page	TDVPS	None	Opaque	N/A	Shared

In addition to the memory operand checks per the table above, the function checks the following conditions (no specific order is implied):

1. RDX must contain a valid RTMR index.

If successful, the function does the following:

2. Extend the RTMR indexed by RDX with the extension data. Extension is done by calculating SHA384 hash over a 96B buffer, composed as follows:
   - Bytes 0 through 47 contain the current RTMR value.
   - Bytes 48 through 95 contain the extension data.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.196: TDG.MR.RTMR.EXTEND Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDG.MR.RTMR.EXTEND is successful.

22.3.5. TDG.VM.RD Leaf

Read a TD-scope metadata field (control structure field) of a TD.

Table 22.197: TDG.VM.RD Input Operands Definition

Operand	Description
RAX	TDCALL instruction leaf and version numbers – see 22.3.1
RCX	Reserved, must be 0
RDX	Field identifier – see 20.9

Table 22.198: TDG.VM.RD Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code – see 22.3.1
R8	Contents of the field In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDG.VM.RD reads a VM-scope metadata field (control structure field) of a TD.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.199 TDG.VM.RD Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)

If the memory operand checks per the table above pass:

1. Read the control structure field.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.200: TDG.VM.RD Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FIELD_NOT_READABLE	
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.

Completion Status Code	Description
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDG.VM.RD is successful.

22.3.6. TDG.VM.WR Leaf

Write a TD-scope metadata field (control structure field) of a TD.

Table 22.201: TDG.VM.WR Input Operands Definition

Operand	Description
RAX	TDCALL instruction leaf and version numbers – see 22.3.1
RCX	Reserved, must be 0
RDX	Field identifier – see 20.9
R8	Data to write to the field
R9	A 64b write mask to indicate which bits of the value in R8 are to be written to the field

Table 22.202: TDG.VM.WR Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code – see 22.3.1
R8	Previous contents of the field In case of an error, as indicated by RAX, R8 returns 0
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDG.VM.WR writes a VM-scope metadata field (control structure field) of a TD.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.203 TDG.VM.WR Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)

If the memory operand checks per the table above pass:

1. Write the control structure field and return its old value.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.204: TDG.VM.WR Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_FIELD_NOT_WRITABLE	

Completion Status Code	Description
TDX_OPERAND_BUSY	Operation encountered a busy operand, indicated by the lower 32 bits of the status. In many cases, this can be resolved by retrying the operation.
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDG.VM.WR is successful.

22.3.7. TDG.VP.CPUIDVE.SET Leaf

TDG.VP.CPUIDVE.SET controls unconditional #VE on CPUID execution by the guest TD.

Table 22.205: TDG.VP.CPUIDVE.SET Input Operands Definition

Operand	Description		
RAX	TDCALL instruction leaf number – see 22.3.1		
RCX	Controls whether CPUID executed by the guest TD will cause #VE unconditionally		
	Bits	**Name**	**Description**
	0	SUPERVISOR	Flags that when CPL is 0, a CPUID executed by the guest TD will cause a #VE unconditionally
	1	USER	Flags that when CPL > 0, a CPUID executed by the guest TD will cause a #VE unconditionally
	63:2	RESERVED	Reserved: must be 0

Table 22.206: TDG.VP.CPUIDVE.SET Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code – see 22.3.1
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

This function controls whether execution of CPUID by the guest TD, when running in supervisor mode and/or in user mode, will unconditionally result in a #VE.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.207 TDG.VP.CPUIDVE.SET Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDVPS structure	TDVPS	RW	Opaque	N/A	Shared

In addition to the memory operand checks per the table above, the function checks the following conditions (no specific order is implied):

1. Reserved bits of RCX must be 0.

If successful, the function does the following:

2. Update the TDVPS.CPUID_VE flags which control unconditional #VE injection for CPUID for the current VCPU.

Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.208: TDG.VP.CPUIDVE.SET Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDG.VP.CPUIDVE.SET is successful.

22.3.8. TDG.VP.INFO Leaf

Get guest TD execution environment information.

Table 22.209: TDG.VP.INFO Input Operands Definition

Operand	Description
RAX	TDCALL instruction leaf number – see 22.3.1

Table 22.210: TDG.VP.INFO Output Operands Definition

Operand	Description		
RAX	TDCALL instruction return code – see 22.3.1 – returns a constant value of TDX_SUCCESS (0)		
RCX	**Bits**	**Name**	**Description**
	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	Reserved: 0
RDX	The TD's ATTRIBUTES (provided as input to TDH.MNG.INIT)		
R8	**Bits**	**Name**	**Description**
	31:0	NUM_VCPUS	Number of Virtual CPUs that are usable (i.e. either active or ready) – see 9.1.3
	63:32	MAX_VCPUS	TD's maximum number of Virtual CPUs (provided as input to TDH.MNG.INIT)
R9	**Bits**	**Name**	**Description**
	31:0	VCPU_INDEX	Virtual CPU index, starting from 0 and allocated sequentially on each successful TDH.VP.INIT
	63:32	RESERVED	Reserved for enumerating future Intel TDX module capabilities, etc.: set to 0
R10	Reserved for enumerating future Intel TDX module capabilities, etc.: set to 0		
R11	Reserved for enumerating future Intel TDX module capabilities, etc.: set to 0		
Other	Unmodified		

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDG.VP.INFO provides the TD software with execution environment information – beyond information that is provided by CPUID.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.211: TDG.VP.INFO Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	R	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDVPS structure	TDVPS	R	Opaque	N/A	Shared

5 ### Completion Status Codes

The table below provides specific notes for status codes returned by this interface function. For a general description of completion status code, see 19.1.

Table 22.212: TDG.VP.INFO Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_SUCCESS	TDG.VP.INFO is successful.

22.3.9. TDG.VP.VEINFO.GET Leaf

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

Get Virtualization Exception Information for the recent #VE exception.

Table 22.213: TDG.VP.VEINFO.GET Input Operands Definition

Operand	Description
RAX	TDCALL instruction leaf number – see 22.3.1

Table 22.214: TDG.VP.VEINFO.GET Output Operands Definition

Operand	Description		
RAX	TDCALL instruction return code – see 22.3.1		
RCX	**Bits**	**Name**	**Description**
	31:0	Exit Reason	The 32-bit value that would have been saved into the VMCS as an exit reason if a VM exit had occurred instead of the virtualization exception
	63:32	Reserved	Reserved: 0
	In case of an error, RCX returns 0		
RDX	Exit Qualification: the 64-bit value that would have been saved into the VMCS as an exit qualification if a legacy VM exit had occurred instead of the virtualization exception In case of an error, as indicated by RAX, RDX returns 0		
R8	Guest Linear Address: the 64-bit value that would have been saved into the VMCS as a guest-linear address if a legacy VM exit had occurred instead of the virtualization exception In case of an error, as indicated by RAX, R8 returns 0		
R9	Guest Physical Address: the 64-bit value that would have been saved into the VMCS as a guest-physical address if a legacy VM exit had occurred instead of the virtualization exception In case of an error, as indicated by RAX, R9 returns 0		
R10	**Bits**	**Name**	**Description**
	31:0	VM-exit instruction length	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
	63:32	VM-exit instruction information	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
	In case of an error, R10 returns 0		
Other	Unmodified		

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDG.VP.VEINFO.GET returns the virtualization exception information of a #VE exception that was previously delivered to the guest TD.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.215: TDG.VP.VEINFO.GET Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	None	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDVPS structure	TDVPS	RW	Opaque	N/A	Shared

The function checks the following conditions (no specific order is implied):

- The VALID field in TDVPS.VE_INFO must non-0 to indicate that a valid virtualization information is available.

If successful, the function does the following:

1. Return the EXIT_REASON, EXIT_QUALIFICATION, GLA, GPA, INSTRUCTION_LENGTH and INSTRUCTION_INFORMATTION from TDVPS.VE_INFO in GPRs.
2. Clear the VALID field in TDVPS.VE_INFO to 0 to indicate that the virtualization information has been read.

Completion Status Codes

Table 22.216: TDG.VP.VEINFO.GET Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_NO_VE_INFO	There is no Virtualization Exception information.
TDX_SUCCESS	TDG.VP.VEINFO.GET is successful.

22.3.10. TDG.VP.VMCALL Leaf

Perform a TD Exit to the host VMM.

Table 22.217: TDG.VP.VMCALL Input Operands Definition

Operand	Description		
RAX	TDCALL instruction leaf number – see 22.3.1		
RCX	A bitmap that controls which part of the guest TD GPR and XMM state is passed as-is to the VMM and back A bit value of 0 indicates that the corresponding register is saved by the Intel TDX module, scrubbed to 0 before SEAMRET to the host VMM, and restored by the Intel TDX module on the following TDH.VP.ENTER. A bit value of 1 indicates that the corresponding register is passed as-is to the host VMM, and on the following TDH.VP.ENTER, the register value is used as input from the host VMM and passed as-is to the guest TD. The value of RCX is passed to the host VMM.		
	Bits	**Name**	**Description**
	15:0	GPR Mask	Controls the transfer of GPR values: Bit 0: RAX (must be 0) Bit 1: RCX (must be 0) Bit 2: RDX Bit 3: RBX Bit 4: RSP (must be 0) Bit 5: RBP Bit 6: RSI Bit 7: RDI Bits 15:8: R15 – R8
	31:16	XMM Mask	Controls the transfer of XMM15 – XMM0 register values
	63:32	Reserved	Reserved: must be 0
RBX, RDX, RBP, RDI, RSI, R8 – R15	If the corresponding bit in RCX is set to 1, the register value passed as-is to the host VMM on SEAMRET. Else, the register value is not used as an input and is preserved.		
XMM0 – XMM15	If the corresponding bit in RCX is set to 1, the register value passed as-is to the host VMM on SEAMRET. Else, the register value is not used as an input and is preserved.		

Table 22.218: TDG.VP.VMCALL Output Operands Definition

Operand	Description
RAX	TDCALL instruction return code: returns a constant value of TDX_SUCCESS (0)
RCX	Unmodified

Operand	Description
RBX, RDX, RBP, RDI, RSI, R8 – R15	If the corresponding bit in RCX is set to 1, the register value passed as-is from the host VMM's SEAMCALL(TDH.VP.ENTER) input. Else, the register value is unmodified.
XMM0 – XMM15	If the corresponding bit in RCX is set to 1, the register value passed as-is from the host VMM's SEAMCALL(TDH.VP.ENTER) input. Else, the register value is unmodified.
Other	Unmodified

Leaf Function Description

Note: The description below is provided at a high level. Actual details, order of checks, returned status codes, etc. may vary.

TDG.VP.VMCALL performs a TD exit to the host VMM. From the VMM's point of view, this is the termination of a previous SEAMCALL(TDH.VP.ENTER). Selected GPR and XMM state is passed to the VMM host, controlled by RCX as shown above. The rest of the CPU state is saved in TDVPS and replaced with a synthetic state.

From the guest TD's point of view, a subsequent SEAMCALL(TDH.VP.ENTER) from the host VMM terminates the TDG.VP.VMCALL function. Most GPR state, and if the value of RCX bit 1 is set, all XMM state, is passed to the TD guest as shown above.

To understand the table and text below, refer to Chapter 17, which explains the general aspects of the Intel TDX interface functions.

Table 22.219: TDG.VP.VMCALL Memory Operands Information Definition

Explicit/ Implicit	Reg.	Addr. Type	Resource	Resource Type	Access	Access Semantics	Align. Check	Concurrency Restrictions
Implicit	N/A	N/A	TDR page	TDR	None	Opaque	N/A	Shared
Implicit	N/A	N/A	TDCS structure	TDCS	None	Opaque	N/A	Shared(i)
Implicit	N/A	N/A	TDVPS structure	TDVPS	R/W	Opaque	N/A	Shared

1. Save guest TD CPU state to TDVPS (including TD VMCS):
   1.1. Save extended state per TDCS.XFAM. There is no strict requirement to save XMM state that will be passed to the host VMM as controlled by RCX. This state will be overwritten on the next TD entry.
   1.2. Save GPR state. There is no strict requirement to save GPR state that will be passed to the host VMM as controlled by RCX (but RCX itself must be saved). This state will be overwritten on the next TD entry.
   1.3. Advance the saved RIP to the instruction following TDCALL.
2. Adjust the TDCS TLB tracking counters.
3. Release the shared locking – acquired on TDH.VP.ENTER of TDR, TDCS and TDVPS.
4. Load host VMM state:
   4.1. Clear the extended state except XMM (per TDCS.XFAM) to synthetic INIT values.
   4.2. As controlled by RCX, either clear or set to the guest TD's value the state of XMM0 – XMM15.
   4.3. As controlled by RCX, either clear or set to the guest TD's value the state of RBX, RDX, RBP, RDI, RSI and R8 – R15.
   4.4. Set RCX to the guest TD's value.
   4.5. Set RAX to the TDCALL exit reason.
   4.6. Restore other host VMM state – saved during TDH.VP.ENTER.
5. Execute SEAMRET to return to the host VMM.

Note: Logically, from the point of view of the guest TD, TDG.VP.VMCALL is terminated by the next TDH.VP.ENTER.

Completion Status Codes

Table 22.220: TDG.VP.VMCALL Completion Status Codes (Returned in RAX) Definition

Completion Status Code	Description
TDX_OPERAND_INVALID	
TDX_SUCCESS	TDG.VP.VMCALL is successful. TD exit was done, resulting a in a completion of SEAMCALL(TDH.VP.ENTER) on the host VMM side. Later, the host VMM executed SEAMCALL(TDH.VP.ENTER) again, and execution returned to the guest TD VCPU (in TDX non-root mode) completing TDG.VP.VMCALL.