Hypervisor Top Level Functional Specification

January, 2020: Released Version 6.0

Abstract

This document is the top-level functional specification (TLFS) of the sixth-generation Microsoft hypervisor. It specifies the hypervisor's externally-visible behavior. The document assumes familiarity with the goals of the project and the high-level hypervisor architecture.

Contents

1 INTRODUCTION 1

1.1 SPECIFICATION STYLE 1

1.2 RESERVED VALUES 1

1.3 REPORT ISSUES 1

1.4 GLOSSARY 1

1.5 SIMPLE SCALAR TYPES 2

1.6 HYPERCALL STATUS CODE 2

1.7 MEMORY ADDRESS SPACE TYPES 2

1.8 STRUCTURES, ENUMERATIONS AND BIT FIELDS 3

1.9 ENDIANNESS 3

1.10 POINTER NAMING CONVENTION 3

2 FEATURE AND INTERFACE DISCOVERY 4

2.1 INTERFACE MECHANISMS 4

2.2 HYPERVISOR DISCOVERY 4

2.3 STANDARD HYPERVISOR CPUID LEAVES 4

2.4 MICROSOFT HYPERVISOR CPUID LEAVES 5

2.4.1 Hypervisor CPUID Leaf Range - 0x40000000 5

2.4.2 Hypervisor Vendor-Neutral Interface Identification - 0x40000001 5

2.4.3 Hypervisor System Identity 6

2.4.4 Hypervisor Feature Identification - 0x40000003 6

2.4.5 Implementation Recommendations - 0x40000004 7

2.4.6 Hypervisor Implementation Limits - 0x40000005 9

2.4.7 Implementation Hardware Features - 0x40000006 9

2.4.8 Hypervisor CPU Management Features - 0x40000007 10

2.4.9 Hypervisor SVM Features - 0x40000008 10

2.4.10 Nested Hypervisor Feature Identification - 0x40000009 11

2.4.11 Hypervisor Nested Virtualization Features - 0x4000000A 12

2.5 VERSIONING 12

2.6 REPORTING THE GUEST OS IDENTITY 13

2.6.1 Proprietary Operating Systems 13

2.6.2 Encoding the Guest OS Identity MSR for Open Source Operating Systems 15

3 HYPERCALL INTERFACE 16

3.1 HYPERCALL OVERVIEW 16

3.2 HYPERCALL CLASSES 16
3.3 HYPERCALL CONTINUATION 17
3.4 HYPERCALL ATOMICITY AND ORDERING 17
3.5 LEGAL HYPERCALL ENVIRONMENTS 17
3.6 ALIGNMENT REQUIREMENTS 18
3.7 HYPERCALL INPUTS 18
3.7.1 Variable Sized Hypercall Input Headers 20
3.7.2 XMM Fast Hypercall Input (formerly "Extended Fast Hypercalls") 21
3.8 HYPERCALL OUTPUTS 21
3.8.1 XMM Fast Hypercall Output 22
3.9 VOLATILE REGISTERS 23
3.10 HYPERCALL DOCUMENTATION 23
3.11 HYPERCALL RESTRICTIONS 24
3.12 HYPERCALL STATUS CODES 25
3.12.1 Output Parameter Validity on Failed Hypercalls 25
3.12.2 Ordering of Error Conditions 25
3.12.3 Common Hypercall Status Codes 25
3.13 ESTABLISHING THE HYPERCALL INTERFACE 26
3.14 EXTENDED HYPERCALL INTERFACE 27
3.14.1 HvExtCallQueryCapabilities 28
4 PARTITION PROPERTIES 30
4.1 OVERVIEW 30
4.2 PARTITION MANAGEMENT DATA TYPES 30
4.2.1 Partition IDs 30
4.2.2 Partition Privilege Flags 30
4.3 PARTITION CRASH ENLIGHTENMENT 34
4.3.1 Guest Crash Enlightenment Interface 34
4.3.2 Guest Crash Enlightenment Data Structure 35
5 GUEST PHYSICAL ADDRESS SPACES 37
5.1 OVERVIEW 37
5.2 PAGE ACCESS RIGHTS 37
5.2.1 GPA Overlay Pages 38
6 HOST INTERCEPTS 40
6.1 OVERVIEW 40
6.1.1 Programmable Intercept Types 40

7 VIRTUAL PROCESSOR MANAGEMENT **42**
7.1 OVERVIEW 42
7.2 VIRTUAL PROCESSOR INDICES 42
7.3 VIRTUAL PROCESSOR REGISTERS 42
7.4 VIRTUAL PROCESSOR STATES 42
7.5 VIRTUAL PROCESSOR IDLE SLEEP STATE 42
7.6 VIRTUAL BOOT PROCESSOR 43
7.7 VIRTUAL PROCESSOR SYNTHETIC MACHINE CHECKS 43
7.8 VIRTUAL PROCESSOR DATA TYPES 43
7.8.1 Virtual Processor Index 43
7.8.2 Virtual Processor Register Names 43
7.8.3 Virtual Processor Register Values 49
7.8.4 Synthetic Machine Check Status Data Structure 50
7.8.5 Synthetic Machine Check Error Code 51
7.8.6 Synthetic Machine Check Event Data Structure 51
7.8.7 Virtual Processor Assist Page 51
7.9 VIRTUAL PROCESSOR REGISTER FORMATS 55
7.9.1 Virtual Processor Run Time Register 55
7.9.2 Virtual Processor Interrupt State Register 55
7.9.3 Virtual Processor Pending Interruption Register 55
7.9.4 Virtual Processor Floating-point and Vector Registers 56
7.9.5 Virtual Processor Segment Registers 58
7.9.6 Virtual Processor Table Registers 59
7.9.7 Synthetic Machine Check Registers 59
7.10 VIRTUAL PROCESSOR INTERFACES 59
7.10.1 HvSetVpRegisters 59
7.10.2 HvGetVpRegisters 62
7.10.3 HvStartVirtualProcessor 65
7.10.4 HvGetVpIndexFromApicId 68
8 VIRTUAL PROCESSOR EXECUTION **71**
8.1 PROCESSOR FEATURES AND CPUID 71
8.2 FAMILY, MODEL AND STEPPING REPORTED BY CPUID 71
8.3 PLATFORM ID REPORTED BY MSR 71
8.4 REAL MODE 71
8.5 MONITOR / MWAIT 72

8.6 System Management Mode 72
8.7 Time Stamp Counter 72
8.8 Memory Accesses 72
8.9 I/O Port Accesses 73
8.10 MSR Accesses 74
8.11 CPUID Execution 75
8.12 Non-Privileged Instruction Execution Prevention (NPIEP) 75
8.13 Exceptions 76
9 VIRTUAL MMU AND CACHING 77
9.1 Virtual MMU Overview 77
9.1.1 Compatibility 77
9.1.2 Legacy TLB Management Operations 78
9.1.3 Virtual TLB Enhancements 78
9.1.4 Restrictions on TLB Flushes 78
9.2 Memory Cache Control Overview 79
9.2.1 Cacheability Settings 79
9.2.2 Mixing Cache Types between a Partition and the Hypervisor 79
9.3 Virtual MMU Data Types 79
9.3.1 Virtual Address Spaces 79
9.3.2 Virtual Address Flush Flags 80
9.3.3 Cache Types 80
9.3.4 Virtual Address Translation Types 80
9.4 Virtual MMU Interfaces 81
9.4.1 HvSwitchVirtualAddressSpace 81
9.4.2 HvFlushVirtualAddressSpace 82
9.4.3 HvFlushVirtualAddressSpaceEx 84
9.4.4 HvFlushVirtualAddressList 85
9.4.5 HvFlushVirtualAddressListEx 88
9.4.6 HvTranslateVirtualAddress 88
9.4.7 HvExtCallGetBootZeroedMemory 92
10 VIRTUAL INTERRUPT CONTROL 94
10.1 Overview 94
10.2 Local APIC 94
10.2.1 Local APIC Virtualization 94
10.2.2 Local APIC Memory-mapped Accesses 95

10.2.3 Local APIC MSR Accesses *95*
10.3 VIRTUAL INTERRUPTS 97
10.3.1 Virtual Interrupt Overview *97*
10.3.2 Virtual Interrupt Types *97*
10.3.3 Trigger Types *97*
10.3.4 EOI Intercepts *98*
10.3.5 EOI Assist *98*
10.4 VIRTUAL INTERRUPT DATA TYPES 99
10.4.1 Interrupt Types *99*
10.4.2 Interrupt Control *99*
10.4.3 Interrupt Vectors *100*
10.4.4 MSI Entry *100*
10.4.5 Interrupt Source *101*
10.4.6 Interrupt Entry *101*
10.4.7 Device Interrupt Target *101*
10.5 VIRTUAL INTERRUPT INTERFACES 101
10.5.1 HvAssertVirtualInterrupt *101*
10.5.2 HvSendSyntheticClusterIpi *104*
10.5.3 HvSendSyntheticClusterIpiEx *105*
10.5.4 HvRetargetDeviceInterrupt *106*
11 INTER-PARTITION COMMUNICATION 109
11.1 OVERVIEW 109
11.2 SYNIC MESSAGES 109
11.3 MESSAGE BUFFERS 109
11.3.1 Guest Message Buffers *109*
11.3.2 Timer Message Buffers *109*
11.3.3 Intercept Message Buffers *109*
11.3.4 Event Log Message Buffers *109*
11.3.5 Message Buffer Queues *109*
11.3.6 Reliability and Sequencing of Guest Message Buffers *109*
11.4 MESSAGES 110
11.4.1 Recommended Message Handling *111*
11.4.2 Message Sources *111*
11.5 SYNIC EVENT FLAGS 112
11.5.1 Event Flag Delivery *112*

11.5.2 Recommended Event Flag Handling 112
11.5.3 Event Flags versus Messages 112
11.6 PORTS AND CONNECTIONS 112
11.7 MONITORED NOTIFICATIONS 113
11.7.1 Monitored Notification Trigger 113
11.7.2 Monitored Notification Latency Hint 113
11.7.3 Monitored Notification Parameters 113
11.7.4 Monitored Notification Page 113
11.8 SYNIC MSRS 114
11.8.1 SCONTROL Register 115
11.8.2 SVERSION Register 115
11.8.3 SIEFP Register 115
11.8.4 SIMP Register 116
11.8.5 SINTx Registers 116
11.8.6 EOM Register 117
11.9 SIM AND SIEF PAGES 117
11.10 INTER-PARTITION COMMUNICATION DATA TYPES 118
11.10.1 Synthetic Interrupt Sources 118
11.10.2 SynIC Message Types 119
11.10.3 SynIC Message Flags 120
11.10.4 SynIC Message Format 120
11.10.5 SynIC Event Flags 121
11.10.6 Ports 122
11.10.7 Connections 123
11.10.8 Connection Information 124
11.10.9 Monitored Notification Page 126
11.11 INTER-PARTITION COMMUNICATION INTERFACES 128
11.11.1 HvPostMessage 128
11.11.2 HvSignalEvent 131
12 TIMERS 133
12.1 OVERVIEW 133
12.1.1 Timer Services 133
12.1.2 Reference Counter 133
12.1.3 Synthetic Timers 133
12.1.4 Periodic Timers 133

12.1.5 Ordering of Timer Expirations 134
12.2 Direct Synthetic Timers 134
12.3 Partition Reference Time Enlightenment 135
12.4 Partition Reference Counter MSR 135
12.4.1 Reference Counter MSR 135
12.5 Synthetic Timer MSRs 135
12.5.1 Synthetic Timer Configuration Register 136
12.5.2 Synthetic Timer Count Register 137
12.6 Synthetic Time-Unhalted Timer MSRs 138
12.6.1 Synthetic Time-Unhalted Timer Configuation MSR 138
12.6.2 Synthetic Time-Unhalted Timer Count MSR 139
12.7 Partition Reference Time Enlightenment 139
12.7.1 Reference Time Stamp Counter (TSC) Page MSR 139
12.7.2 Format of the Reference TSC Page 140
12.7.3 Partition Reference TSC Mechanism 140
13 MESSAGE FORMATS 142
13.1 Overview 142
13.2 Message Data Types 142
13.2.1 Message Header 142
13.2.2 Intercept Message Header 142
13.2.3 VP Execution State 143
13.2.4 I/O Port Access Information 143
13.2.5 Exception Information 144
13.2.6 Memory Access Flags 144
13.3 Timer Messages 144
13.3.1 Timer Expiration Message 144
14 SCHEDULER 146
14.1 Scheduling Concepts 146
14.2 Scheduling Policy Settings 146
14.2.1 CPU Reserve 146
14.2.2 CPU Cap 146
14.2.3 CPU Weight 147
14.3 Other Scheduling Considerations 147
14.3.1 Guest Spinlocks 147
14.4 Scheduler Data Types 147

14.5 SCHEDULER INTERFACES 148

14.5.1 HvNotifyLongSpinWait 148

15 VIRTUAL SECURE MODE 150

15.1 OVERVIEW 150

15.2 VIRTUAL TRUST LEVELS (VTL) 150

15.2.1 VTL Protections 151

15.3 VSM DETECTION, ENABLING, AND STATUS 151

15.3.1 VSM Detection 151

15.3.2 VSM Capability Detection 151

15.3.3 VSM Status Registers 152

15.3.4 Partition VTL Initial state 153

15.4 VTL ENABLEMENT 153

15.4.1 Enabling a Target VTL for a Partition 153

15.4.2 Enabling a Target VTL for Virtual Processors 153

15.5 VTL CONFIGURATION 153

15.5.1 Partition Configuration 154

15.5.2 Configuring Lower VTLs 155

15.6 VTL ENTRY 155

15.6.1 VTL Call 156

15.7 VTL EXIT 157

15.7.1 VTL Return 157

15.7.2 Restrictions 157

15.8 HYPERCALL PAGE ASSISTS 158

15.8.1 VTL Control via the VP Assist Page 158

15.8.2 Definition 158

15.9 MEMORY ACCESS PROTECTIONS 159

15.9.1 Memory Protection Hierarchy 159

15.9.2 Memory Access Violations 160

15.9.3 Default memory protection types 160

15.10 MODE BASED EXECUTION CONTROL (MBEC) 160

15.10.1 Descriptor Tables 161

15.10.2 MBEC configuration 161

15.10.3 MBEC Interaction with Supervisor Mode Execution Prevention (SMEP) 161

15.11 VIRTUAL PROCESSOR STATE ISOLATION 162

15.11.1 Private State 162

15.11.2 Shared State ... *163*
15.11.3 Real Mode ... *164*
15.12 VTL Interrupt Management ... 164
15.12.1 Overview ... *164*
15.12.2 Interrupts targeted at a higher VTL ... *164*
15.12.3 Interrupts Targeted at a Lower VTL ... *164*
15.12.4 Virtual Interrupt Notification Assist ... *165*
15.13 Secure Intercepts ... 165
15.13.1 Secure Intercept Types ... *166*
15.13.2 Nested Intercepts ... *166*
15.13.3 Handling Secure Intercepts ... *166*
15.13.4 Secure Register Intercepts ... *166*
15.14 DMA and Devices ... 167
15.15 VSM Interfaces ... 167
15.15.1 Type Definitions ... *167*
15.15.2 HvModifyVtlProtectionMask ... *168*
15.15.3 HvEnablePartitionVtl ... *169*
15.15.4 HvEnableVpVtl ... *171*
15.15.5 HvVtlCall ... *175*
15.15.6 HvVtlReturn ... *176*
16 NESTED VIRTUALIZATION ... 177
16.1 Overview ... 177
16.2 Definitions ... 177
16.3 Requirements ... 177
16.3.1 Guest Partition ... *177*
16.3.2 Supported Platforms ... *177*
16.4 Guest Enlightenments ... 178
16.4.1 Enlightened Interface Discovery ... *178*
16.5 Enlightened VMCS ... 178
16.5.1 Enlightened VMCS Versioning ... *178*
16.5.2 Clean Fields ... *178*
16.5.3 Enlightened MSR Bitmap ... *179*
16.6 Compatibility with Live Migration ... 179
16.6.1 Live Migration Notifications ... *179*
16.6.2 TSC Emulation ... *179*

16.7 VIRTUAL TLB 180
16.8 DIRECT VIRTUAL FLUSH 180
16.8.1 Configuration 180
16.8.2 Partition Assist Page 181
16.8.3 Synthetic VM-Exit 181
16.9 SECOND LEVEL ADDRESS TRANSLATION 181
16.9.1 Compatibility 181
16.10 NESTED MSR ACCESS RESTRICTION 182
16.11 NESTED VIRTUALIZATION DATA TYPES 183
16.11.1 GPA Range 183
16.11.2 Enlightened VMCS 183
16.11.3 Clean Fields 186
16.11.4 Physical VMCS Encoding 186
16.12 NESTED VIRTUALIZATION EXCEPTIONS 189
16.13 NESTED VIRTUALIZATION INTERFACES 190
16.13.1 HvFlushGuestPhysicalAddressSpace 190
16.13.2 HvFlushGuestPhysicalAddressList 191
16.14 NESTED VIRTUALIZATION MSRS 192
16.14.1 Nested VP Index Register 192
16.14.2 Nested SynIC MSRs 192
17 APPENDIX A: HYPERCALL CODE REFERENCE 198
18 APPENDIX B: HYPERCALL STATUS CODE REFERENCE 203
19 APPENDIX C: HYPERVISOR SYNTHETIC MSRS 206

1 Introduction

This document is the top-level functional specification (TLFS) of the sixth-generation Microsoft hypervisor. It specifies the externally-visible behavior to guest partitions. The document assumes familiarity with the goals of the project and the high-level hypervisor architecture.

This document is intended to be sufficiently complete and precise to allow a developer to implement a guest partition interface with the Microsoft hypervisor.

This document is not intended to document the interface between the hypervisor and the root partition virtualization stack. For more information on that interface, please reference the Windows Hypervisor Platform documentation: https://aka.ms/WindowsHypervisorPlatformDocsTLFS.

1.1 Specification Style

This specification is informal; that is, the interfaces are not specified in a formal language. Nevertheless, it is a goal to be precise. It is also a goal to specify which behaviors are architectural and which are implementation-specific. Callers should not rely on behaviors that fall into the latter category because they may change in future implementations.

Segments of code and algorithms are presented with a grey background.

1.2 Reserved Values

This specification documents some fields as "reserved." These fields may be given specific meaning in future versions of the hypervisor architecture. For maximum forward compatibility, clients of the hypervisor interface should follow the guidance provided within this document. In general, two forms of guidance are provided.

Preserve value (documented as *RsvdP* in diagrams and `ReservedP` in code segments) **–** For maximum forward compatibility, clients should preserve the value within this field. This is typically done by reading the current value, modifying the values of the non-reserved fields, and writing the value back.

Zero value (documented as *RsvdZ* in diagrams and `ReservedZ` in code segments) **–** For maximum forward compatibility, clients should zero the value within this field.

Reserved fields within read-only structures are simply documented as *Rsvd* in diagrams and simply as `Reserved` in code segments. For maximum forward compatibility, the values within these fields should be ignored. Clients should not assume these values will always be zero.

1.3 Report Issues

If you notice errors in this document, or would like to give feedback, please file an issue in the Hyper-V Documentation GitHub repository: https://aka.ms/VirtualizationDocumentationIssuesTLFS.

1.4 Glossary

Partition – Hyper-V supports isolation in terms of a partition. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute.

Root Partition – The root partition (a.k.a the "parent" or "host") is a privileged management partition. The root partition manages machine-level functions such as device drivers, power management, and device addition/removal. The virtualization stack runs in the parent partition and has direct access to the hardware devices. The root partition then creates the child partitions which host the guest operating systems.

Child Partition – The child partition (a.k.a. the "guest") hosts a guest operating system. All access to physical memory and devices by a child partition is provided via the Virtual Machine Bus (VMBus) or the hypervisor.

Hypercall – Hypercalls are an interface for communication with the hypervisor.

1.5 Simple Scalar Types

Hypervisor data types are built up from simple scalar types UINT8, UINT16, UINT32, UINT64 and UINT128. Each of these represents a simple unsigned integer scalar with the specified bit count. Several corresponding signed integer scalars are also defined: INT8, INT16, INT32, and INT64.

The hypervisor uses neither floating point instructions nor floating point types.

1.6 Hypercall Status Code

Every hypercall returns a 16-bit status code of type HV_STATUS.

```
typedef UINT16 HV_STATUS;
```

All hypercall status codes are documented in Appendix B.

1.7 Memory Address Space Types

The hypervisor architecture defines three independent address spaces:

- *System physical addresses* (SPAs) define the physical address space of the underlying hardware as seen by the CPUs. There is only one system physical address space for the entire machine.
- *Guest physical addresses* (GPAs) define the guest's view of physical memory. GPAs can be mapped to underlying SPAs. There is one guest physical address space per partition.
- *Guest virtual addresses* (GVAs) are used within the guest when it enables address translation and provides a valid guest page table.

All three of these address spaces are up to 2^{64} bytes in size. The following types are thus defined:

```
typedef UINT64 HV_SPA;
typedef UINT64 HV_GPA;
typedef UINT64 HV_GVA;
```

Many hypervisor interfaces act on pages of memory rather than single bytes. The minimum page size is architecture-dependent. For x64, it is defined as 4K.

```
#define X64_PAGE_SIZE 0x1000

#define HV_X64_MAX_PAGE_NUMBER (MAXUINT64/X64_PAGE_SIZE)
#define HV_PAGE_SIZE X64_PAGE_SIZE
#define HV_LARGE_PAGE_SIZE X64_LARGE_PAGE_SIZE
#define HV_PAGE_MASK (HV_PAGE_SIZE - 1)

typedef UINT64 HV_SPA_PAGE_NUMBER;
typedef UINT64 HV_GPA_PAGE_NUMBER;
typedef UINT64 HV_GVA_PAGE_NUMBER;

typedef UINT32 HV_SPA_PAGE_OFFSET

typedef HV_GPA_PAGE_NUMBER *PHV_GPA_PAGE_NUMBER;
```

To convert an HV_SPA to an HV_SPA_PAGE_NUMBER, simply divide by HV_PAGE_SIZE.

1.8 Structures, Enumerations and Bit Fields

Many data structures and constant values defined later in this specification are defined in terms of C-style enumerations and structures. The C language purposely avoids defining certain implementation details. However, this document assumes the following:

- All enumerations declared with the "enum" keyword define 32-bit signed integer values.
- All structures are padded in such a way that fields are aligned naturally (that is, an 8-byte field is aligned to an offset of 8 bytes and so on).
- All bit fields are packed from low-order to high-order bits with no padding.

1.9 Endianness

The hypervisor interface is designed to be endian-neutral (that is, it should be possible to port the hypervisor to a big-endian or little-endian system), but some of the data structures defined later in this specification assume little-endian layout. Such data structures will need to be amended if and when a big-endian port is attempted.

1.10 Pointer Naming Convention

The document uses a naming convention for pointer types. In particular, a "P" prepended to a defined type indicates a pointer to that type. A "PC" prepended to a defined type indicates a pointer to a constant value of that type.

2 Feature and Interface Discovery

2.1 Interface Mechanisms

Guest software interacts with the hypervisor through a variety of mechanisms. Many of these mirror the traditional mechanisms used by software to interact with the underlying processor. As such, these mechanisms are architecture-specific. On the x64 architecture, the following mechanisms are used:

- **CPUID instruction –** Used for static feature and version information.
- **MSRs (model-specific registers) –** Used for status and control values.
- **Memory-mapped registers –** Used for status and control values.
- **Processor interrupts –** Used for asynchronous events, notifications and messages.

In addition to these architecture-specific interfaces, the hypervisor provides a simple procedural interface implemented with hypercalls. For information about the hypercall mechanism, see chapter 3.

2.2 Hypervisor Discovery

Before using any hypervisor interfaces, software should first determine whether it is running within a virtualized environment. On x64 platforms that conform to this specification, this is done by executing the CPUID instruction with an input (EAX) value of 1. Upon execution, code should check bit 31 of register ECX (the "hypervisor present bit"). If this bit is set, a hypervisor is present. In a non-virtualized environment, the bit will be clear.

```
CPUID.01h.ECX:31        // if set, virtualization present
```

If the "hypervisor present bit" is set, additional CPUID leaves can be queried for more information about the conformant hypervisor and its capabilities. Two such leaves are guaranteed to be available: 0x40000000 and 0x40000001. Subsequently-numbered leaves may also be available.

2.3 Standard Hypervisor CPUID Leaves

When the leaf at 0x40000000 is queried, the hypervisor will return information that provides the maximum hypervisor CPUID leaf number and a vendor ID signature.

Leaf	Information Provided	
0x40000000	EAX	The maximum input value for hypervisor CPUID information.
	EBX	Hypervisor Vendor ID Signature
	ECX	Hypervisor Vendor ID Signature
	EDX	Hypervisor Vendor ID Signature

If the leaf at 0x40000001 is queried, it will return a value representing a vendor-neutral hypervisor interface identification. This determines the semantics of the leaves from 0x4000002 through 0x400000FF.

Leaf	**Information Provided**	
0x40000001	EAX	Hypervisor Interface Signature
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

These two leaves allow the guest to query the hypervisor vendor ID and interface independently. The vendor ID is provided only for informational and diagnostic purposes. It is recommended that software only base compatibility decisions on the interface signature reported through leaf 0x40000001.

2.4 Microsoft Hypervisor CPUID Leaves

On hypervisors conforming to the Microsoft hypervisor CPUID interface, the 0x40000000 and 0x40000001 leaf registers will have the following values.

2.4.1 **Hypervisor CPUID Leaf Range - 0x40000000**

EAX determines the maximum hypervisor CPUID leaf. EBX-EDX contain the hypervisor vendor ID signature. The vendor ID signature should be used only for reporting and diagnostic purposes.

Leaf	Information Provided	
0x40000000	EAX	The maximum input value for hypervisor CPUID information. On Microsoft hypervisors, this will be at least 0x40000005.
	EBX	0x7263694D—“Micr”
	ECX	0x666F736F—“osof”
	EDX	0x76482074—“t Hv”

2.4.2 **Hypervisor Vendor-Neutral Interface Identification - 0x40000001**

EAX contains the hypervisor interface identification signature. This determines the semantics of the leaves from 0x40000002 through 0x400000FF.

Leaf	**Information Provided**	
0x40000001	EAX	0x31237648—"Hv#1"
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

Hypervisors conforming to the "Hv#1" interface also provide at least the following leaves.

2.4.3 **Hypervisor System Identity**

This value will be zero until the OS identity MSR is set (see section 2.6); after that, it has the following definitions:

Leaf	**Information Provided**	
0x40000002	EAX	Build Number
	EBX	Bits 31-16: Major Version Bits 15-0: Minor Version
	ECX	Service Pack
	EDX	Bits 31-24: Service Branch Bits 23-0: Service Number

2.4.4 **Hypervisor Feature Identification - 0x40000003**

EAX and EBX indicate which features are available to the partition based upon the current partition privileges.

Leaf	Information Provided	
0x40000003	EAX	Bits 31-0: Corresponds to bits 31-0 of HV_PARTITION_PRIVILEGE_MASK (see 4.2.2 Partition Privilege Flags)
	EBX	Bits 31-0: Corresponds to bits 63-32 of HV_PARTITION_PRIVILEGE_MASK (see 4.2.2 Partition Privilege Flags)
	ECX	Bits 31-0: Reserved
	EDX	Bit 0: Deprecated (previously indicated availability of the MWAIT instruction). Bit 1: Guest debugging support is available. Bit 2: Performance Monitor support is available.

		Bit 3: Support for physical CPU dynamic partitioning events is available. Bit 4: Support for passing hypercall input parameter block via XMM registers is available. Bit 5: Support for a virtual guest idle state is available. Bit 6: Support for hypervisor sleep state is available. Bit 7: Support for querying NUMA distances is available. Bit 8: Support for determining timer frequencies is available. Bit 9: Support for injecting synthetic machine checks is available. Bit 10: Support for guest crash MSRs is available. Bit 11: Support for debug MSRs is available. Bit 12: Support for NPIEP is available. Bit 13: DisableHypervisorAvailable Bit 14: ExtendedGvaRangesForFlushVirtualAddressListAvailable Bit 15: Support for returning hypercall output via XMM registers is available. Bit 16: Reserved Bit 17: SintPollingModeAvailable Bit 18: HypercallMsrLockAvailable Bit 19: Use direct synthetic timers Bit 20: Support for PAT register available for VSM Bit 21: Support for bndcfgs register available for VSM Bit 22: Reserved Bit 23: Support for synthetic time unhalted timer available Bits 25-24: Reserved Bit 26: Use of Intel's Last Branch Record (LBR) feature is only supported if this bit is set Bits 31-27: Reserved

2.4.5 **Implementation Recommendations - 0x40000004**

Indicates which behaviors the hypervisor recommends the OS implement for optimal performance.

Leaf	Information Provided	
0x40000004	EAX	Bit 0: Recommend using hypercall for address space switches rather than MOV to CR3 instruction. Bit 1: Recommend using hypercall for local TLB flushes rather than INVLPG or MOV to CR3 instructions. Bit 2: Recommend using hypercall for remote TLB flushes rather than inter-processor interrupts. Bit 3: Recommend using MSRs for accessing APIC registers EOI, ICR and TPR rather than their memory-mapped

		counterparts. Bit 4: Recommend using the hypervisor-provided MSR to initiate a system RESET. Bit 5: Recommend using relaxed timing for this partition. If used, the VM should disable any watchdog timeouts that rely on the timely delivery of external interrupts. Bit 6: Recommend using DMA remapping. Bit 7: Recommend using interrupt remapping. Bit 8: Reserved Bit 9: Recommend deprecating AutoEOI. Bit 10: Recommend using SyntheticClusterIpi hypercall. Bit 11: Recommend using the newer ExProcessorMasks interface. Bit 12: Indicates that the hypervisor is nested within a Hyper-V partition. Bit 13: Recommend using INT for MBEC system calls Bit 14: Recommend a nested hypervisor using the enlightened VMCS interface. Also indicates that additional nested enlightenments may be available (see leaf 0x4000000A) Bit 15: UseSyncedTimeline – Indicates the partition should consume the QueryPerformanceCounter bias provided by the root partition. Bit 16: Reserved Bit 17: UseDirectLocalFlushEntire – Indicates the guest should toggle CR4.PGE to flush the entire TLB, as this is more performant than making a hypercall. Bit 18: NoNonArchitecturalCoreSharing - indicates that core sharing is not possible. This can be used as an optimization to avoid the performance overhead of STIBP. Bit 31-19: Reserved
	EBX	Recommended number of attempts to retry a spinlock failure before notifying the hypervisor about the failures. 0xFFFFFFFF indicates never notify.
	ECX	Bits 6-0: ImplementedPhysicalAddressBits – Reports the physical address width (MAXPHYADDR) reported by the system’s physical processors. If all bits contain 0, the feature is not supported. Note that the value reported is the actual number of physical address bits, and not the bit position used to represent that number. Bits 31-7: Reserved

	EDX	Reserved

2.4.6 **Hypervisor Implementation Limits - 0x40000005**

Describes the scale limits supported in the current hypervisor implementation. If any value is zero, the hypervisor does not expose the corresponding information; otherwise, they have these meanings.

Leaf	Information Provided	
0x40000005	EAX	The maximum number of virtual processors supported.
	EBX	The maximum number of logical processors supported.
	ECX	The maximum number of physical interrupt vectors available for interrupt remapping.
	EDX	Reserved

2.4.7 **Implementation Hardware Features - 0x40000006**

Indicates which hardware-specific features have been detected and are currently in use by the hypervisor.

Leaf	Information Provided	
0x40000006	EAX	Bit 0: Support for APIC overlay assist is detected and in use. Bit 1: Support for MSR bitmaps is detected and in use. Bit 2: Support for architectural performance counters is detected and in use. Bit 3: Support for second level address translation is detected and in use. Bit 4: Support for DMA remapping is detected and in use. Bit 5: Support for interrupt remapping is detected and in use. Bit 6: Indicates that a memory patrol scrubber is present in the hardware. Bit 7: DMA protection is in use. Bit 8: HPET is requested. Bit 9: Synthetic timers are volatile. Bits 31-10: Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

2.4.8 **Hypervisor CPU Management Features - 0x40000007**

Indicates enlightenments that are available to the root partition only.

Leaf	Information Provided	
0x40000007	EAX	Bit 0: StartLogicalProcessor Bit 1: CreateRootvirtualProcessor Bit 2: PerformanceCounterSync Bits 30-3: Reserved Bit 31: ReservedIdentityBit
	EBX	Bit 0: ProcessorPowerManagement Bit 1: MwaitIdleStates Bit 2: LogicalProcessorIdling Bits 31-3: Reserved
	ECX	Bit 0: RemapGuestUncached Bit 31-1: Reserved
	EDX	Reserved

2.4.9 **Hypervisor SVM Features - 0x40000008**

Indicates support for shared virtual memory (SVM).

Leaf	Information Provided	
0x40000008		
	EAX	Bit 0: SvmSupported Bits 10-1: Reserved0 Bits 31-11: MaxPasidSpacePasidCount
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

2.4.10 **Nested Hypervisor Feature Identification - 0x40000009**

Describes the features exposed to the partition by the hypervisor when running nested. EAX describes access to virtual MSRs. EDX describes access to hypercalls.

Leaf	Information Provided	
0x40000009	EAX	Bits 1-0: Reserved Bit 2: AccessSynicRegs Bit 3: Reserved Bit 4: AccessIntrCtrlRegs Bit 5: AccessHypercallMsrs Bit 6: AccessVpIndex Bits 11-7: Reserved Bit 12: AccessReenlightenmentControls Bits 31-13: Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	Bits 3-0: Reserved Bit 4: XmmRegistersForFastHypercallAvailable Bits 14-5: Reserved Bit 15: FastHypercallOutputAvailable Bit 16: Reserved Bit 17: SintPollingModeAvailable Bits 31:18 Reserved

2.4.11 **Hypervisor Nested Virtualization Features - 0x4000000A**

Indicates which nested virtualization optimizations are available to a nested hypervisor.

Leaf	Information Provided	
0x4000000A	EAX	Bits 7-0: Enlightened VMCS version (low) Bits 15-8: Enlightened VMCS version (high) Bit 16: Reserved Bit 17: Indicates support for direct virtual flush hypercalls. Bit 18 Indicates support for the HvFlushGuestPhysicalAddressSpace and HvFlushGuestPhysicalAddressList hypercalls. Bit 19: Indicates support for using an enlightened MSR bitmap. Bit 20: Indicates support for combining virtualization exceptions in the page fault exception class Bits 31-21: Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

2.5 Versioning

The hypervisor version information is encoded in leaf 0x40000002. Two version numbers are provided: the main version and the service version.

The main version includes a major and minor version number and a build number. These correspond to Microsoft Windows release numbers. The service version describes changes made to the main version. For maximum forward compatibility, clients should use the hypervisor version information with extreme care. When checking main versions, clients should use greater-than-or-equal tests, not equality tests. The following pseudo-code demonstrates the method that should be employed when comparing entire version numbers (consisting of both the main and service versions):

```
if <your-main-version> greater than <hypervisor-main-version>
   {
   your version is compatible
   }
else if <your-main-version> equal to <hypervisor-main-version>
   and
      <your-service-version> greater than or equal to
                                    <hypervisor-service-version>
   {
   your version is compatible
   }
else
   {
   your version is NOT compatible
   }
```

Clients are strongly encouraged to check for hypervisor features by using CPUID leaves 0x40000003 through 0x40000005 rather than by comparing against version ranges.

2.6 Reporting the Guest OS Identity

The guest OS running within the partition must identify itself to the hypervisor by writing its signature and version to an MSR (HV_X64_MSR_GUEST_OS_ID). This MSR is partition-wide and is shared among all virtual processors (virtual processors are described in chapter 7, Virtual Processor Management).

This register's value is initially zero. A non-zero value must be written to the Guest OS ID MSR before the hypercall code page can be enabled (see Establishing the Hypercall Interface). If this register is subsequently zeroed, the hypercall code page will be disabled.

```
#define HV_X64_MSR_GUEST_OS_ID 0x40000000
```

2.6.1 **Proprietary Operating Systems**

The following is the recommended encoding for this MSR. Some fields may not apply for some guest OSs.

63	62:48	47:40	39:32	31:24	23:16	15:0
OS Type	Vendor ID	OS ID	Major Version	Minor Version	Service Version	Build Number

Bits	Description	Attributes
63	OS Type Indicates the OS type. A value of 0 indicates a proprietary, closed source OS. A value of 1 indicates an open source OS.	Read/Write
62:48	Vendor ID Indicates the guest OS vendor. A value of 0 is reserved. A list of known vendors is listed in the next section.	Read/write
47:40	OS ID Indicates the OS variant. Encoding is unique to the vendor. Microsoft operating systems are encoded as follows: 0=Undefined, 1=MS-DOS®, 2=Windows® 3.x, 3=Windows® 9x, 4=Windows® NT (and derivatives), 5=Windows® CE	Read/write
39:32	Major Version Indicates the major version of the OS	Read/write
31:24	Minor Version Indicates the minor version of the OS	Read/write
23:16	Service Version Indicates the service version (for example, "service pack" number)	Read/write
15:0	Build Number Indicates the build number of the OS	Read/write

2.6.1.1 **Known Vendors**

Vendor values are allocated by Microsoft. To request a new vendor, please file an issue on the GitHub virtualization documentation repository: https://aka.ms/VirtualizationDocumentationIssuesTLFS

Vendor	Value
Microsoft	0x0001
HPE	0x0002
LANCOM	0x0200

2.6.2 Encoding the Guest OS Identity MSR for Open Source Operating Systems

The following encoding is offered as guidance for open source operating system vendors intending to conform to this specification. It is suggested that open source operating systems adopt the following convention.

Bits	Description	Attributes
63	Open Source Bit 63 should be set to 1 to indicate an Open Source OS.	Read/write
62:56	OS Type Bits 62-56 should specify the OS type (e.g., Linux, FreeBSD, etc.). A full list of known OS Types is captured in the next section.	Read/write
55:48	OS ID Bits 55:48 may specify any additional vendor information (distribution-specific identification).	Read/write
47:16	Version Bits 47:16 should specify the upstream kernel version information.	Read/write
15:0	Build Number Bits 15:0 should specify any additional identification.	Read/write

2.6.2.1 Known OS Types

OS Type values are allocated by Microsoft. To request a new OS Type, please file an issue on the GitHub virtualization documentation repository: https://aka.ms/VirtualizationDocumentationIssuesTLFS

OS Type	Value
Linux	0x1
FreeBSD	0x2
Xen	0x3
Illumos	0x4

3 Hypercall Interface

3.1 Hypercall Overview

The hypervisor provides a calling mechanism for guests. Such calls are referred to as *hypercalls*. Each hypercall defines a set of input and/or output parameters. These parameters are specified in terms of a memory-based data structure. All elements of the input and output data structures are padded to natural boundaries up to 8 bytes (that is, two-byte elements must be on two-byte boundaries and so on).

A second hypercall calling convention can optionally be used for a subset of hypercalls – in particular, those that have two or fewer input parameters and no output parameters. When using this calling convention, the input parameters are passed in registers.

A third hypercall calling convention can optionally be used for a subset of hypercalls where the input parameter block is up to 112 bytes. When using this calling convention, the input parameters are passed in registers, including the volatile XMM registers.

Input and output data structures must both be placed in memory on an 8-byte boundary and padded to a multiple of 8 bytes in size. The values within the padding regions are ignored by the hypervisor.

For output, the hypervisor is allowed to (but not guaranteed to) overwrite padding regions. If it overwrites padding regions, it will write zeros.

3.2 Hypercall Classes

There are two classes of hypercalls: *simple* and *rep* (short for "repeat"). A *simple hypercall* performs a single operation and has a fixed-size set of input and output parameters. A *rep hypercall* acts like a series of simple hypercalls. In addition to a fixed-size set of input and output parameters, rep hypercalls involve a list of fixed-size input and/or output elements.

When a caller initially invokes a rep hypercall, it specifies a *rep count* that indicates the number of elements in the input and/or output parameter list. Callers also specify a *rep start index* that indicates the next input and/or output element that should be consumed. The hypervisor processes rep parameters in list order – that is, by increasing element index.

For subsequent invocations of the rep hypercall, the *rep start index* indicates how many elements have been completed – and, in conjunction with the *rep count* value – how many elements are left. For example, if a caller specifies a rep count of 25, and only 20 iterations are completed within the 50µs window (described in section 3.3), the hypercall returns control back to the calling virtual processor after updating the *rep start index* to 20. (See section 3.7 for more information about the *rep start index*.) When the hypercall is re-executed, the hypervisor will resume at element 20 and complete the remaining 5 elements.

If an error is encountered when processing an element, an appropriate status code is provided along with a *reps completed count*, indicating the number of elements that were successfully processed before the error was encountered. Assuming the specified hypercall control word is valid (see the following) and the input / output parameter lists are accessible, the hypervisor is guaranteed to attempt at least one rep, but it is not required to process the entire list before returning control back to the caller.

3.3 Hypercall Continuation

A hypercall can be thought of as a complex instruction that takes many cycles. The hypervisor attempts to limit hypercall execution to 50μs or less before returning control to the virtual processor that invoked the hypercall. Some hypercall operations are sufficiently complex that a 50μs guarantee is difficult to make. The hypervisor therefore relies on a *hypercall continuation* mechanism for some hypercalls – including all rep hypercall forms.

The hypercall continuation mechanism is mostly transparent to the caller. If a hypercall is not able to complete within the prescribed time limit, control is returned back to the caller, but the instruction pointer is not advanced past the instruction that invoked the hypercall. This allows pending interrupts to be handled and other virtual processors to be scheduled. When the original calling thread resumes execution, it will re-execute the hypercall instruction and make forward progress toward completing the operation.

Most simple hypercalls are guaranteed to complete within the prescribed time limit. However, a small number of simple hypercalls might require more time. These hypercalls use hypercall continuation in a similar manner to rep hypercalls. In such cases, the operation involves two or more internal states. The first invocation places the object (for example, the partition or virtual processor) into one state, and after repeated invocations, the state finally transitions to a terminal state. For each hypercall that follows this pattern, the visible side effects of intermediate internal states is described.

3.4 Hypercall Atomicity and Ordering

Except where noted, the action performed by a hypercall is atomic both with respect to all other guest operations (for example, instructions executed within a guest) and all other hypercalls being executed on the system. A simple hypercall performs a single atomic action; a rep hypercall performs multiple, independent atomic actions.

Simple hypercalls that use hypercall continuation may involve multiple internal states that are externally visible. Such calls comprise multiple atomic operations.

Each hypercall action may read input parameters and/or write results. The inputs to each action can be read at any granularity and at any time after the hypercall is made and before the action is executed. The results (that is, the output parameters) associated with each action may be written at any granularity and at any time after the action is executed and before the hypercall returns.

The guest must avoid the examination and/or manipulation of any input or output parameters related to an executing hypercall. While a virtual processor executing a hypercall will be incapable of doing so (as its guest execution is suspended until the hypercall returns), there is nothing to prevent other virtual processors from doing so. Guests behaving in this manner may crash or cause corruption within their partition.

3.5 Legal Hypercall Environments

Hypercalls can be invoked only from the most privileged guest processor mode. In the case of x64, this means protected mode with a current privilege level (CPL) of zero. Although real-mode code runs with an effective CPL of zero, hypercalls are *not* allowed in real mode. An attempt to invoke a hypercall within an illegal processor mode will generate a #UD (undefined operation) exception.

All hypercalls should be invoked through the architecturally-defined hypercall interface. (See the following sections for instructions on discovering and establishing this interface.) An attempt to invoke a hypercall by any other means (for example, copying the code from the hypercall code page to an alternate location and executing it from there) *might* result in an undefined operation (#UD) exception. The hypervisor is not guaranteed to deliver this exception.

3.6 Alignment Requirements

Callers must specify the 64-bit guest physical address (GPA) of the input and/or output parameters. GPA pointers must by 8-byte aligned. If the hypercall involves no input or output parameters, the hypervisor ignores the corresponding GPA pointer.

The input and output parameter lists cannot overlap or cross page boundaries. Hypercall input and output pages are expected to be GPA pages and not "overlay" pages (for a discussion of overlay pages, see section 5.2.1). If the virtual processor writes the input parameters to an overlay page and specifies a GPA within this page, hypervisor access to the input parameter list is undefined.

The hypervisor will validate that the calling partition can read from the input page before executing the requested hypercall. This validation consists of two checks: the specified GPA is mapped and the GPA is marked *readable*. If either of these tests fails, the hypervisor generates a memory intercept message. For more information on memory intercepts, see Chapter 11.

For hypercalls that have output parameters, the hypervisor will validate that the partition can write to the output page. This validation consists of two checks: the specified GPA is mapped and the GPA is marked *writable*. If either of these tests fails, the hypervisor attempts to generate a memory intercept message. If the validation succeeds, the hypervisor "locks" the output GPA for the duration of the operation. Any attempt to remap or unmap this GPA will be deferred until after the hypercall is complete.

3.7 Hypercall Inputs

Callers specify a hypercall by a 64-bit value called a *hypercall input value*. It is formatted as follows:

63:60	**59:48**	**47:44**	**43:32**	**31:27**	**26**	**25:17**	**16**	**15:0**
RsvdZ (4 bits)	Rep start index (12 bits)	RsvdZ (4 bits)	Rep count (12 bits)	RsvdZ (5 bits)	Is Nested (1 bit)	Variable header size (9 bits)	Fast (1 bit)	Call Code (16 bits)

Call code	16 bits	Specifies which hypercall is requested
Fast	1 bit	Specifies whether the hypercall uses the register-based calling convention. 0: Use the memory-based calling convention 1: Use the register-based calling convention
Variable header size	9 bits	The size of a variable header, in the number of 64-bit values in the variable-sized header. See 3.7.1.
RsvdZ	5 bits	Must be zero
Is Nested	1 bit	Specifies the hypercall should be handled by the L0 hypervisor in a nested environment. 0: The call should be handled by a guest hypervisor 1: The call should be handled by the L0 hypervisor
Rep Count	12 bits	Total number of reps (for rep call, must be zero otherwise)
RsvdZ	4 bits	Must be zero
Rep Start Index	12 bits	Starting index (for rep call, must be zero otherwise)
RsvdZ	4 bits	Must be zero

For rep hypercalls, the *rep count* field indicates the total number of reps. The *rep start index* indicates the particular repetition relative to the start of the list (zero indicates that the first element in the list is to be processed). Therefore, the *rep count* value must always be greater than the *rep start index*.

Register mapping for hypercall inputs when the Fast flag is zero:

x64	x86	Contents
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameters **GPA**
R8	EDI:ESI	Output Parameters **GPA**

The hypercall input value is passed in registers along with a GPA that points to the input and output parameters. The register mappings depend on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode. The hypervisor determines the caller's mode based on the value of EFER.LMA and CS.L. If both of these flags are set, the caller is assumed to be a 64-bit caller.

Register mapping for hypercall inputs when the Fast flag is one:

x64	x86	Contents
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameter
R8	EDI:ESI	Input Parameter

The hypercall input value is passed in registers along with the input parameters. The register mappings depend on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode. The hypervisor determines the caller's mode based on the value of EFER.LMA and CS.L. If both of these flags are set, the caller is assumed to be a 64-bit caller.

This mapping is not universal. When XMM Fast Hypercall Input is supported, please refer to section 3.7.2.2 instead.

3.7.1 **Variable Sized Hypercall Input Headers**

Most hypercall input *headers* have fixed size. The amount of header data being passed from the guest to the hypervisor is therefore implicitly specified by the hypercall code and need not be specified separately. However, some hypercalls require a variable amount of header data. These hypercalls typically have a fixed size input header and additional header input that is of variable size.

A variable sized header is similar to a fixed hypercall input (aligned to 8 bytes and sized to a multiple of 8 bytes). The caller must specify how much total data it is providing as input headers. This size is provided as part of the hypercall input value (see "Variable header size" in table above).

Since the fixed header size is implicit, instead of supplying the total header size, only the variable portion is supplied in the input controls:

$$Variable\ Header\ Bytes\ = \{Total\ Header\ Bytes\ - \\ sizeof\ (Fixed\ Header)\}\ rounded\ up\ to\ nearest\ multiple\ of\ 8$$

$$Variable\ Header\ Size\ = \frac{Variable\ Header\ Bytes}{8}$$

It is illegal to specify a non-zero variable header size for a hypercall that is not explicitly documented as accepting variable sized input headers. In such a case the hypercall will result in a return code of HV_STATUS_INVALID_HYPERCALL_INPUT.

It is possible that for a given invocation of a hypercall that does accept variable sized input headers that all the header input fits entirely within the fixed size header. In such cases the variable sized input header is zero-sized and the corresponding bits in the hypercall input should be set to zero.

In all other regards, hypercalls accepting variable sized input headers are otherwise similar to fixed size input header hypercalls with regards to calling conventions. It is also possible for a variable sized header hypercall to additionally support rep semantics. In such a case the rep elements lie after the header in the usual fashion, except that the header's total size includes both the fixed and variable portions. All other rules remain the same e.g. the first rep element must be 8 byte aligned.

3.7.2 XMM Fast Hypercall Input (formerly "Extended Fast Hypercalls")

The hypervisor supports the use of *XMM fast hypercalls*, which allows some hypercalls to take advantage of the improved performance of the fast hypercall interface even though they require more than two input parameters. The XMM fast hypercall interface uses six XMM registers to allow the caller to pass an input parameter block up to 112 bytes in size.

3.7.2.1 Feature Enumeration

Availability of the XMM fast hypercall interface is indicated via the "Hypervisor Feature Identification" CPUID Leaf (0x40000003, see section 2.4.4):

- Bit 4: support for passing hypercall input via XMM registers is available.

Note that there is a separate flag to indicate support for XMM fast output. Any attempt to use this interface when the hypervisor does not indicate availability will result in a #UD fault.

3.7.2.2 Register Mapping – Input Only

For hypervisor versions with XMM fast input support, callers can use the following register mapping:

x64	x86	Contents
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameter Block
R8	EDI:ESI	Input Parameter Block
XMM0	XMM0	Input Parameter Block
XMM1	XMM1	Input Parameter Block
XMM2	XMM2	Input Parameter Block
XMM3	XMM3	Input Parameter Block
XMM4	XMM4	Input Parameter Block
XMM5	XMM5	Input Parameter Block

The hypercall input value is passed in registers along with the input parameters. The register mappings depend on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode. The hypervisor determines the caller's mode based on the value of EFER.LMA and CS.L. If both of these flags are set, the caller is assumed to be a 64-bit caller. If the input parameter block is smaller than 112 bytes, any extra bytes in the registers are ignored.

3.8 Hypercall Outputs

All hypercalls return a 64-bit value called a *hypercall result value*. It is formatted as follows:

63:44	43:32	31:16	15:0
Rsvd (20 bits)	Reps complete (12 bits)	Rsvd (16 bits)	Result (16 bits)

Result	16 bits	HV_STATUS code indicating success or failure
Rsvd	16 bits	Callers should ignore the value in these bits
Reps completed	12 bits	Number of reps successfully completed
Rsvd	20 bits	Callers should ignore the value in these bits

For rep hypercalls, the *reps complete* field is the total number of reps complete and not relative to the *rep start index*. For example, if the caller specified a *rep start index* of 5, and a *rep count* of 10, the *reps complete* field would indicate 10 upon successful completion.

The hypercall result value is passed back in registers. The register mapping depends on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode (see above). The register mapping for hypercall outputs is as follows:

x64	X86	Content
RAX	EDX:EAX	Hypercall Result Value

3.8.1 XMM Fast Hypercall Output

Similar to how the hypervisor supports XMM fast hypercall inputs, the same registers can be shared to return output. This is only supported on x64 platforms.

3.8.1.1 Feature Enumeration

The ability to return output via XMM registers is indicated via the “Hypervisor Feature Identification” CPUID Leaf (0x40000003, see section 2.4.4):

- Bit 15: support for returning hypercall output via XMM registers is available.

Note that there is a separate flag to indicate support for XMM fast input. Any attempt to use this interface when the hypervisor does not indicate availability will result in a #UD fault.

3.8.1.2 Register Mapping – Input and Output

Registers that are not being used to pass input parameters can be used to return output. In other words with this capability, if the input parameter block is smaller than 112 bytes (rounded up to the nearest 16 byte aligned chunk), the remaining registers will return hypercall output.

x64	Contents
RDX	Input or Output Block
R8	Input or Output Block
XMM0	Input or Output Block
XMM1	Input or Output Block
XMM2	Input or Output Block
XMM3	Input or Output Block
XMM4	Input or Output Block
XMM5	Input or Output Block

For example, if the input parameter block is 20 bytes in size, the hypervisor would ignore the following 12 bytes. The remaining 80 bytes would contain hypercall output (if applicable).

3.9 Volatile Registers

Hypercalls will only modify the specified register values under the following conditions:

1. RAX (x64) and EDX:EAX (x86) are always overwritten with the hypercall result value and output parameters, if any (discussed in section 3.8).
2. Rep hypercalls will modify RCX (x64) and EDX:EAX (x86) with the new rep start index.
3. HvSetVpRegisters can modify any registers that are supported with that hypercall (see section 7.10.1).
4. RDX, R8, and XMM0 through XMM5, when used for fast hypercall **input**, remain unmodified. However, registers used for fast hypercall **output** can be modified, including RDX, R8, and XMM0 through XMM5 (see 3.8.1). Hyper-V will only modify these registers for fast hypercall output, which is limited to x64.

3.10 Hypercall Documentation

Each hypercall in this document is described in two ways: a *wrapper interface* and a *native interface*. The wrapper interface is the recommended high-level (C-style) calling convention typically provided by a "wrapper library" that runs within the guest (for example, *winhv.sys* on Microsoft Windows®). The native interface is the one actually provided by the hypervisor.

The recommended wrapper interface is described using standard C-style notation. The following is an example of a wrapper interface for the hypothetical *HvAssignWidgets* hypercall:

```
HV_STATUS
HvAssignWidgets(
   __in    HV_PARTITION_ID PartitionId,
   __in    UINT64        Flags,
   __inout PUINT32       RepCount,
__in    PCHV_WIDGET     WidgetList
);
```

The native interface is defined in terms of memory-based data structures. Up to four data structures may be defined:

- Input parameter header
- Input list element (for rep hypercalls)
- Output parameter header
- Output list element (for rep hypercalls)

The following is an example of the native interface documentation for the hypothetical *HvAssignWidgets* hypercall:

HvAssignWidgets [rep]		
	Call Code = 0xBADD	
➡ InputParameter Header		
0	PartitionId (8 bytes)	
8	Flags (8 bytes)	
➡ Input List Element		
0	WidgetId (8 bytes)	
8	WidgetType (4 bytes)	Padding (4 bytes)

The above is an example of a rep (repeating) hypercall. As input, it has two fixed parameters and an input list consisting of one or more elements. The first list element can be found at offset 16. The list element is described using offsets within the element itself, starting with 0.

3.11 Hypercall Restrictions

Hypercalls may have restrictions associated with them for them to perform their intended function. If all restrictions are not met, the hypercall will terminate with an appropriate error. The following restrictions will be listed, if any apply:

- The calling partition must possess a particular privilege (see 4.2.2 for information regarding privilege flags)
- The partition being acted upon must be in a particular state (e.g. “Active”)
- The partition must be the root
- The partition must be either a parent or child

- The virtual processor must be in a particular state (see section 7.4 for information regarding virtual processor states).

3.12 Hypercall Status Codes

Each hypercall is documented as returning an output value that contains several fields. A status value field (of type HV_STATUS) is used to indicate whether the call succeeded or failed. The hypercall status value field is discussed in section 3.7.1.

3.12.1 Output Parameter Validity on Failed Hypercalls

Unless explicitly stated otherwise, when a hypercall fails (that is, the result field of the hypercall result value contains a value other than HV_STATUS_SUCCESS), the content of all output parameters are indeterminate and should not be examined by the caller. Only when the hypercall succeeds, will all appropriate output parameters contain valid, expected results.

3.12.2 Ordering of Error Conditions

Error conditions are not presented in this document in any particular sequence. The order in which error conditions are detected and reported by the hypervisor is undefined. In other words, if multiple errors exist, the hypervisor must choose which error condition to report. Priority should be given to those error codes offering greater security, the intent being to prevent the hypervisor from revealing information to callers lacking sufficient privilege. For example, the status code HV_STATUS_ACCESS_DENIED is the preferred status code over one that would reveal some context or state information purely based upon privilege.

3.12.3 Common Hypercall Status Codes

Several result codes are common to all hypercalls and are therefore not documented for each hypercall individually. These include the following:

Status code	Error condition
HV_STATUS_SUCCESS	The call succeeded.
HV_STATUS_INVALID_HYPERCALL_CODE	The hypercall code is not recognized.
HV_STATUS_INVALID_HYPERCALL_INPUT	The rep count is incorrect (for example, a non-zero rep count is passed to a non-rep call or a zero rep count is passed to a rep call).
	The rep start index is not less than the rep count.
	A reserved bit in the specified hypercall input value is non-zero.
HV_STATUS_INVALID_ALIGNMENT	The specified input or output GPA pointer is not aligned to 8 bytes.

Status code	Error condition
	The specified input or output parameter lists spans pages.
	The input or output GPA pointer is not within the bounds of the GPA space.

The return code HV_STATUS_SUCCESS indicates that no error condition was detected. A full list of hypercall status codes can be found in Appendix B.

3.13 Establishing the Hypercall Interface

Hypercalls are invoked by using a special opcode. Because this opcode differs among virtualization implementations, it is necessary for the hypervisor to abstract this difference. This is done through a special *hypercall page*. This page is provided by the hypervisor and appears within the guest's GPA space. The guest is required to specify the location of the page by programming the Guest Hypercall MSR.

```
#define HV_X64_MSR_HYPERCALL 0x40000001
```

Bits	Description	Attributes
63:12	Hypercall GPFN Indicates the **Guest Physical Page Number** of the hypercall page	Read/write
11:2	RsvdP Guest should ignore on reads and preserve on writes	Reserved
1	Locked Indicates if this MSR is immutable. If set, this MSR is locked, thereby preventing the relocation of the hypercall page. Once set, only system reset can clear this bit.	Read/Write (unless set)
0	Enable Hypercall Page Enables the hypercall page	Read/write

The hypercall page can be placed anywhere within the guest's GPA space, but must be page-aligned. If the guest attempts to move the hypercall page beyond the bounds of the GPA space, a #GP fault will result when the MSR is written.

This MSR is a partition-wide MSR. In other words, it is shared by all virtual processors in the partition. If one virtual processor successfully writes to the MSR, another virtual processor will read the same value.

Before the hypercall page is enabled, the guest OS must report its identity by writing its version signature to a separate MSR (HV_X64_MSR_GUEST_OS_ID). If no guest OS identity has been specified, attempts to enable the hypercall will fail. The enable bit will remain zero even if a one is written to it. Furthermore, if the guest OS identity is cleared to zero after the hypercall page has been enabled, it will become disabled.

The hypercall page appears as an “overlay” to the GPA space; that is, it covers whatever else is mapped to the GPA range. Its contents are readable and executable by the guest. Attempts to write to the hypercall page will result in a protection (#GP) exception.

After the hypercall page has been enabled, invoking a hypercall simply involves a call to the start of the page.

The following is a detailed list of the steps involved in establishing the hypercall page:

1. The guest reads CPUID leaf 1 and determines whether a hypervisor is present by checking bit 31 of register ECX.
2. The guest reads CPUID leaf 0x40000000 to determine the maximum hypervisor CPUID leaf (returned in register EAX) and CPUID leaf 0x40000001 to determine the interface signature (returned in register EAX). It verifies that the maximum leaf value is at least 0x40000005 and that the interface signature is equal to “Hv#1”. This signature implies that HV_X64_MSR_GUEST_OS_ID, HV_X64_MSR_HYPERCALL and HV_X64_MSR_VP_INDEX are implemented.
3. The guest writes its OS identity into the MSR HV_X64_MSR_GUEST_OS_ID if that register is zero.
4. The guest reads the Hypercall MSR (HV_X64_MSR_HYPERCALL).
5. The guest checks the Enable Hypercall Page bit. If it is set, the interface is already active, and steps 6 and 7 should be omitted.
6. The guest finds a page within its GPA space, preferably one that is not occupied by RAM, MMIO, and so on. If the page is occupied, the guest should avoid using the underlying page for other purposes.
7. The guest writes a new value to the Hypercall MSR (HV_X64_MSR_HYPERCALL) that includes the GPA from step 6 and sets the Enable Hypercall Page bit to enable the interface.
8. The guest creates an executable VA mapping to the hypercall page GPA.
9. The guest consults CPUID leaf 0x40000003 to determine which hypervisor facilities are available to it.

After the interface has been established, the guest can initiate a hypercall. To do so, it populates the registers per the hypercall protocol and issues a CALL to the beginning of the hypercall page. The guest should assume the hypercall page performs the equivalent of a near return (0xC3) to return to the caller. As such, the hypercall must be invoked with a valid stack.

3.14 Extended Hypercall Interface

Hypercalls with call codes above 0x8000 are known as *extended hypercalls*. Extended hypercalls use the same calling convention as normal hypercalls and appear identical from a guest VM’s perspective. Extended hypercalls are internally handled differently within the Hyper-V hypervisor.

Below is a list of extended hypercalls.

Extended Hypercall Name	**Call Code**
HvExtCallQueryCapabilities	0x8001
HvExtCallGetBootZeroedMemory	0x8002
HvExtCallMemoryHeatHint	0x8003
HvExtCallEpfSetup	0x8004
HvExtCallSchedulerAssistSetup	0x8005
HvExtCallMemoryHeatHintAsync	0x8006

Extended hypercall capabilities can be queried with HvExtCallQueryCapabilities. The availability of HvExtCallQueryCapabilities is reported as a partition privilege flag (see 4.2.2).

3.14.1 **HvExtCallQueryCapabilities**

This hypercall reports the availability of extended hypercalls.

Wrapper Interface

```
HV_STATUS
HvExtCallQueryCapabilities(
      __out  UINT64          Capabilities;
      );
```

Native Interface

HvExtCallQueryCapabilities	
	Call Code = 0x8001
⬅ Output Parameters	
0	Capabilities (8 bytes)

Input Parameters

None.

Output Parameters

Capabilities – the extended hypercalls supported by the hypervisor. A value of “1” indicates that the extended hypercall is available.

Bits	Extended Hypercall	Call Code
0	HvExtCallGetBootZeroedMemory	0x8002

Bits	Extended Hypercall	Call Code
1	HvExtCallMemoryHeatHint	0x8003
2	HvExtCallEpfSetup	0x8004
3	HvExtCallSchedulerAssistSetup	0x8005
4	HvExtCallMemoryHeatHintAsync	0x8006
63:5	Reserved	

Restrictions

- The availability of this hypercall must be queried using the EnableExtendedHypercalls partition privilege flag.

4 Partition Properties

4.1 Overview

This section describes how partition privileges and capabilities are defined.

4.2 Partition Management Data Types

4.2.1 Partition IDs

Partitions are identified by using a partition ID. This 64-bit number is allocated by the hypervisor. All partitions are guaranteed by the hypervisor to have unique IDs. Note that these are not "globally unique" in that the same ID may be generated across a power cycle (that is, a reboot of the hypervisor). However, the hypervisor guarantees that IDs created within a single power cycle are unique.

```
typedef UINT64 HV_PARTITION_ID;
typedef HV_PARTITION_ID *PHV_PARTITION_ID;
```

The guest should not ascribe any meaning to the value of a partition ID. The "invalid" partition ID is used in several interfaces to indicate an invalid partition.

```
#define HV_PARTITION_ID_INVALID     ((HV_PARTITION_ID) 0x0)
```

A partition can specify its own ID using HV_PARTITION_ID_SELF

```
#define HV_PARTITION_ID_SELF    ((HV_PARTITION_ID) -1)
```

4.2.2 Partition Privilege Flags

Each partition has a set of properties that are assigned to it by the hypervisor.

One of the partition properties (HvPartitionPropertyPrivilegeFlags) defines the hypervisor facilities that the partition is allowed to access. This enables the parent to control which synthetic MSRs and hypercalls a guest partition can access.

The property is defined with the following structure. Reserved fields are set to 0 to ensure forward compatibility:

```
typedef struct
{
// Access to virtual MSRs
UINT64  AccessVpRunTimeReg:1;
UINT64  AccessPartitionReferenceCounter:1;
UINT64  AccessSynicRegs:1;
UINT64  AccessSyntheticTimerRegs:1;
UINT64  AccessIntrCtrlRegs:1;
UINT64  AccessHypercallMsrs:1;
UINT64  AccessVpIndex:1;
UINT64  AccessResetReg:1;
UINT64  AccessStatsReg:1;
UINT64  AccessPartitionReferenceTsc:1;
UINT64  AccessGuestIdleReg:1;
UINT64  AccessFrequencyRegs:1;
UINT64  AccessDebugRegs:1;
UINT64  AccessReenlightenmentControls:1
UINT64  Reserved1:18;

// Access to hypercalls
UINT64  CreatePartitions:1;
UINT64  AccessPartitionId:1;
UINT64  AccessMemoryPool:1;
UINT64  Reserved:1;
UINT64  PostMessages:1;
UINT64  SignalEvents:1;
UINT64  CreatePort:1;
UINT64  ConnectPort:1;
UINT64  AccessStats:1;
UINT64  Reserved2:2;
UINT64  Debugging:1;
UINT64  CpuManagement:1;
UINT64  Reserved:1
UINT64  Reserved:1;
UINT64  Reserved:1;
UINT64  AccessVSM:1;
UINT64  AccessVpRegisters:1;
UINT64  Reserved:1;
UINT64  Reserved:1;
UINT64  EnableExtendedHypercalls:1;
UINT64  StartVirtualProcessor:1;
UINT64  Reserved3:10;
} HV_PARTITION_PRIVILEGE_MASK;
```

The following table explains what each of these flags controls.

Privilege Flag	Meaning
AccessVpRunTimeReg	The partition has access to the synthetic MSR HV_X64_MSR_VP_RUNTIME. If this flag is cleared, accesses to this MSR results in a #GP fault if the MSR intercept is not installed.
AccessPartitionReferenceCounter	The partition has access to the partition-wide reference count MSR, HV_X64_MSR_TIME_REF_COUNT. If this flag is cleared, accesses to this MSR results in a #GP fault if the MSR intercept is not installed.
AccessSynicRegs	The partition has access to the synthetic MSRs associated with the Synic (HV_X64_MSR_SCONTROL through HV_X64_MSR_EOM and HV_X64_MSR_SINT0 through HV_X64_MSR_SINT15). If this flag is cleared, accesses to these MSRs results in a #GP fault if the MSR intercept is not installed.
AccessSyntheticTimerMsrs	The partition has access to the synthetic MSRs associated with the Synic (HV_X64_MSR_STIMER0_CONFIG through HV_X64_MSR_STIMER3_COUNT). If this flag is cleared, accesses to these MSRs results in a #GP fault if the MSR intercept is not installed.
AccessIntrCtrlRegs	The partition has access to the synthetic MSRs associated with the APIC (HV_X64_MSR_EOI, HV_X64_MSR_ICR and HV_X64_MSR_TPR). If this flag is cleared, accesses to these MSRs results in a #GP fault if the MSR intercept is not installed.
AccessHypercallMsrs	The partition has access to the synthetic MSRs related to the hypercall interface (HV_X64_MSR_GUEST_OS_ID and HV_X64_MSR_HYPERCALL). If this flag is cleared, accesses to these MSRs result in a #GP fault if the MSR intercept is not installed.
AccessVpIndex	The partition has access to the synthetic MSR that returns the virtual processor index. If this flag is cleared, accesses to this MSR results in a #GP fault if the MSR intercept is not installed.

Privilege Flag	Meaning
AccessResetReg	This partition has access to the synthetic MSR that resets the system. If this flag is cleared, accesses to this MSR results in a #GP fault if the MSR intercept is not installed.
AccessStatsReg	This partition has access to the synthetic MSRs that allows the guest to map and unmap its own statistics pages.
AccessPartitionReferenceTsc	The partition has access to the reference TSC.
AccessGuestIdleReg	The partition has access to the synthetic MSR that allows the guest to enter the guest idle state.
AccessFrequencyRegs	The partition has access to the synthetic MSRs that supply the TSC and APIC frequencies, if supported.
AccessDebugRegs	The partition has access to the synthetic MSRs used for some forms of guest debugging.
AccessReenlightenmentControls	The partition has access to reenlightenment controls. See chapter 16 for more details.
CreatePartitions	The partition can invoke the hypercall HvCreatePartition. The partition also can make any other hypercall that is restricted to operating on children.
AccessPartitionId	The partition can invoke the hypercall HvGetPartitionId to obtain its own partition ID.
AccessMemoryPool	The partition can invoke the hypercalls HvDepositMemory, HvWithdrawMemory and HvGetMemoryBalance.
PostMessages	The partition can invoke the hypercall HvPostMessage.
SignalEvents	The partition can invoke the hypercall HvSignalEvent.
CreatePort	The partition can invoke the hypercall HvCreatePort.
ConnectPort	The partition can invoke the hypercall HvConnectPort.
AccessStats	The partition can invoke the hypercalls HvMapStatsPage and HvUnmapStatsPage.

Privilege Flag	Meaning
Debugging	The partition can invoke the hypercalls HvPostDebugData, HvRetrieveDebugData and HvResetDebugSession.
CpuManagement[1]	The partition can invoke the hypercalls HvGetLogicalProcessorRunTime and HvCallParkedVirtualProcessors. This partition also has access to the power management MSRs.
AccessVSM	The partition can use VSM.
AccessVpRegisters	The partition can invoke the hypercalls HvSetVpRegisters and HvGetVpRegisters.
EnableExtendedHypercalls	The partition can use the extended hypercall interface. Callers must query for extended hypercall capabilities using HvExtCallQueryCapabilities. See Extended Hypercall Interface.
StartVirtualPRocessor	The partition can use HvStartVirtualProcessor to initialize virtual processors.

4.3 Partition Crash Enlightenment

The hypervisor provides guest partitions with a crash enlightenment facility. This interface allows the operating system running in a guest partition the option of providing forensic information about fatal OS conditions to the hypervisor as part of its crash dump procedure. Options include preserving the contents of the guest crash parameter MSRs and specifying a crash message. The hypervisor then makes this information available to the root partition for logging. This mechanism allows the virtualization host administrator to gather information about the guest OS crash event without needing to inspect persistent storage attached to the guest partition for crash dump or core dump information that may be stored there by the crashing guest OS.

The availability of this mechanism is indicated via CPUID.0x400003.EDX:10, the GuestCrashMsrsAvailable flag; refer to Hypervisor Feature Identification.

4.3.1 Guest Crash Enlightenment Interface

The guest crash enlightenment interface is provided through six synthetic MSRs, as defined below.

[1] Some implementations may restrict this partition privilege to the root partition.

```
#define HV_X64_MSR_CRASH_P0                        0x40000100
#define HV_X64_MSR_CRASH_P1                        0x40000101
#define HV_X64_MSR_CRASH_P2                        0x40000102
#define HV_X64_MSR_CRASH_P3                        0x40000103
#define HV_X64_MSR_CRASH_P4                        0x40000104

#define HV_X64_MSR_CRASH_CTL                       0x40000105
```

4.3.1.1 Guest Crash Control MSR

The guest crash control MSR HV_X64_MSR_CRASH_CTL may be used by guest partitions to determine the hypervisor's guest crash capabilities, and to invoke the specified action to take.

Determining Guest Crash Capabilities

To determine the guest crash capabilities, guest partitions may read the HV_X64_MSR_CRASH_CTL register. The supported set of actions and capabilities supported by the hypervisor is reported.

Invoking Guest Crash Capabilities

To invoke a supported hypervisor guest crash action, a guest partition writes to the HV_X64_MSR_CRASH_CTL register, specifying the desired action. Two variations are supported: CrashNotify by itself, and CrashMessage in combination with CrashNotify. For each occurrence of a guest crash, at most a single write to MSR HV_X64_MSR_CRASH_CTL should be performed, specifying one of the two variations.

Guest Crash Action	Description
CrashMessage	This action is used in combination with CrashNotify to specify a crash message to the hypervisor. When selected, the values of P3 and P4 are treated as the location and size of the message. HV_X64_MSR_CRASH_P3 is the guest physical address of the message, and HV_X64_MSR_CRASH_P4 is the length in bytes of the message (maximum of 4096 bytes).
CrashNotify	This action indicates to the hypervisor that the guest partition has completed writing the desired data into the guest crash parameter MSRs (i.e., P0 thru P4), and the hypervisor should proceed with logging the contents of these MSRs.

4.3.2 Guest Crash Enlightenment Data Structure

The following data structure is used to define the contents of the guest crash enlightenment control register, _HV_CRASH_CTL_REG_CONTENTS.

```
typedef union _HV_CRASH_CTL_REG_CONTENTS
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Reserved     : 62;   // Reserved bits
        UINT64 CrashMessage : 1;    // P3 is the PA of the message
                                    // P4 is the length in bytes
        UINT64 CrashNotify  : 1;    // Log contents of crash parameter
                                    // system register
    };
} HV_CRASH_CTL_REG_CONTENTS;
```

5 Guest Physical Address Spaces

5.1 Overview

The size of the GPA space for a partition is the range from 0 to some maximum address that depends on architectural attributes of the virtual machine exposed by the partition.

Each page within a GPA space is in one of three states:

- *Mapped*: A mapped GPA page is associated with a RAM SPA page.
- *Inaccessible:* An inaccessible GPA page may not be read, written, or executed by the partition.
- *Unmapped*: An unmapped GPA page is not associated with a RAM SPA page.

For guest partitions:

- Its GPA mappings are not necessarily identity-mapped. That is, a GPA does not necessarily refer to the same SPA.
- The GPA mappings are defined by the partition's parent. At the time they are mapped, they are specified in terms of the parent's GPA space. Therefore, these pages must be mapped into the parent's GPA space; however, the parent is not required to have read, write or execute access to these mapped pages.
- When a virtual processor accesses an unmapped GPA page, the hypervisor suspends the virtual processor and sends a message to the partition's parent. Code within the parent will typically respond by creating a mapping or by emulating the instruction that generated the memory access. In either case, it is up to the software in the parent partition to "unsuspend" the child's virtual processor.

5.2 Page Access Rights

Mapped GPA pages have the following attributes which define the access rights of the partition:

- *Readable*: Data on the page can be read.
- *Writeable*: Data to the page can be written.
- *Executable*: Code on the page can be executed.

These access rights are enforced for explicit accesses performed by the guest's virtual processors. They are also enforced for implicit reads or writes performed by the hypervisor (for example, due to guest page table flag updates).

Access right combinations are limited by the underlying hardware. The following table shows the valid combinations for an x64 system.

Access Type			Description
Read	Write	Exec	
•	•	•	Instruction fetches, reads, and writes are allowed
	•	•	Illegal combination

Access Type			Description
Read	Write	Exec	
●		●	Instruction fetches and reads are allowed
		●	Illegal combination
●	●		Reads and writes are allowed
	●		Illegal combination
●			Reads are allowed
			No access is allowed

If an attempted memory access is not permitted according to the access rights, the virtual processor that performed the access is suspended (on an instruction boundary) and a message is sent to the parent partition. Code within the parent will typically respond by adjusting the access rights to allow the access or emulating the instruction that performed the memory access. In either case, it is up to the software in the parent partition to “unsuspend” the child’s virtual processor.

Memory accesses that cross page boundaries are handled in a manner that is consistent with the underlying processor architecture. For x64, this means the entire access is validated before any data exchange occurs. For example, if a four-byte write is split across two pages and the first page is writable but the second is not, the first two bytes are not written.

5.2.1 GPA Overlay Pages

The hypervisor defines several special pages that “overlay” the guest’s GPA space. The hypercall code page is an example of an overlay page. Overlays are addressed by guest physical addresses but are not included in the normal GPA map maintained internally by the hypervisor. Conceptually, they exist in a separate map that overlays the GPA map.

If a page within the GPA space is overlaid, any SPA page mapped to the GPA page is effectively “obscured” and generally unreachable by the virtual processor through processor memory accesses. Furthermore, access rights installed on the underlying GPA page are not honored when accessing an overlay page.

If an overlay page is disabled or is moved to a new location in the GPA space, the underlying GPA page is “uncovered”, and an existing mapping becomes accessible to the guest.

If multiple overlay pages are programmed to appear on top of each other (for example, the guest programs the APIC to appear on top of the hypercall page), the hypervisor will choose an ordering (which is undefined) and only one of these overlays will be visible to code running within the partition. In such cases, if the “top-most” overlay is disabled or moved, another overlay page will become visible.

When the hypervisor performs a guest page table walk, it might find that a page table is located on a GPA location associated with an overlay page. In this case, the hypervisor may choose to do any one of the following: generate a guest page fault, reference the contents of the overlay page, or reference the contents of the underlying GPA mapping. Because this behavior can vary from one hypervisor implementation to the next, it is strongly recommended that guests avoid this situation.

6 Host Intercepts

6.1 Overview

This section describes the principal mechanism the hypervisor provides to facilitate the virtualization of certain guest events. These events occur when a virtual processor executes certain instructions or generates certain exceptions. An authorized guest (a parent partition) can install an *intercept* for certain events on another guest (a child partition, or lesser privileged VTL). An intercept involves the detection of an event performed by a virtual processor (explicitly or implicitly). When an intercepted event occurs in the child partition, the virtual processor that triggered the event is suspended, and an *intercept message* is sent to the parent. The virtual processor remains suspended until the parent explicitly clears the virtual processor register.

In general, the register state of the virtual processor when it is suspended corresponds to the state before the execution of the instruction that triggered the intercept. As such, the instruction can be restarted.

The purpose of this mechanism is to allow a virtualization-aware parent to create a virtual environment that allows an unmodified legacy guest—that was written to execute on the physical hardware—to execute in a hypervisor partition. Such legacy guests may attempt to access physical devices that do not exist in a hypervisor partition (for example, by accessing certain I/O ports). The mechanism described in this section makes it possible to intercept all such accesses and transfer control to the parent partition. The parent partition can alter the effect of the intercepted instruction such that, to the child, it mirrors the expected behavior in physical hardware.

An intercept only affects the state of a single virtual processor. Other virtual processors within the same partition continue to run. Therefore, it's possible that multiple intercept messages can be "in progress" concurrently. Intercept messages are queued to the parent in the order in which they are detected.

6.1.1 **Programmable Intercept Types**

The available processor intercept events depend on the (virtual) processor architecture and the capabilities of the physical hardware's virtualization facilities.

The following types of processor events can be intercepted on x64 platforms:

- Accesses to I/O Ports
- Accesses to MSRs
- Execution of the CPUID instruction
- Exceptions
- Accesses to registers
- Hypercalls

Additional intercept types were added as part of the Windows Hypervisor Platform API. For documentation on those intercept types, reference https://aka.ms/WindowsHypervisorPlatformDocsTLFS.

The following table describes the scope and intercept access flags that are allowed for each intercept type:

Intercept Type	Intercept Applies To	Valid Access Flags
I/O port access (see section 8.9)	A specific I/O port.	Read *and* Write access flags must be specified to install an intercept.
MSR access (see section 8.10)	All MSRs not being virtualized by the hypervisor. Note that certain privileges affect MSR virtualization.	Read *and* Write access flags must be specified to install the intercept.
CPUID instruction execution (see section 8.11)	A specific CPUID leaf.	Execute access flag must be specified to install an intercept.
Exceptions	A specific exception vector.	Execute access flag must be specified to install an intercept.
Control Register Access	A specific control register.	Read or Write access flags must be specified to install the intercept.

7 Virtual Processor Management

7.1 Overview

Each partition may have zero or more virtual processors. This section describes virtual processor state and how it is managed.

7.2 Virtual Processor Indices

A virtual processor is identified by a tuple composed of its partition ID and its processor index. The processor index is assigned to the virtual processor when it is created, and it is unchanged through the lifetime of the virtual processor. Processor indices are described in 7.8.1.

7.3 Virtual Processor Registers

Associated with each virtual processor is a variety of state modeled as processor registers. Most of this state is defined by the underlying processor architecture and consists of architected register values. The hypervisor provides a mechanism for reading and writing these registers through hypercalls HvGetVpRegisters and HvSetVpRegisters.

If a virtual processor register is modified and the newly-specified value is invalid in some way, the hypervisor may or may not immediately return an error. In some cases, a value is invalid only in certain contexts (for example, if a bit within another virtual processor register is set). Therefore, some invalid register values are not detected until the virtual processor resumes execution. In such a case, the virtual processor is suspended, and an intercept message (with a message type HvMessageTypeInvalidVpRegisterValue) is sent to its parent partition.

7.4 Virtual Processor States

Conceptually, a virtual processor is in one of four states:

- *Running*—actively consuming processor cycles of a logical processor
- *Ready*—ready to run, but not actively running because other virtual processors are running
- *Waiting*—in a state defined by the processor architecture that does not involve the active execution of instructions (for example, for the x64 architecture, at a HLT instruction, within “waiting for SIPI” state or if the scheduler has capped the virtual processor)
- *Suspended*—stopped on a guest instruction boundary either explicitly suspended or implicitly suspended due to an intercept. Both suspension reasons must be cleared before a virtual processor becomes active.

7.5 Virtual Processor Idle Sleep State

Virtual processors may be placed in a virtual idle processor power state, or processor sleep state. This enhanced virtual idle state allows a virtual processor that is placed into a low power idle state to be woken with the arrival of an interrupt even when the interrupt is masked on the virtual processor. In other words, the virtual idle state allows the operating system in the guest partition to take advantage of processor power saving techniques in the OS that would otherwise be unavailable when running in a guest partition.

A partition which possesses the AccessGuestIdleMsr privilege (refer to section 4.2.2) may trigger entry into the virtual processor idle sleep state through a read to the hypervisor-defined MSR

HV_X64_MSR_GUEST_IDLE. The virtual processor will be woken when an interrupt arrives, regardless of whether the interrupt is enabled on the virtual processor or not.

7.6 Virtual Boot Processor

The virtual processor created with the index of zero is the *virtual boot processor* for the partition that it is related to. It will be the only virtual processor with the BSP flag set in the IA32_APIC_BASE_MSR register. Virtual processors created with non-zero indices are *virtual application processors.* Both the virtual boot processor and virtual application processors may be created or deleted at any time.

7.7 Virtual Processor Synthetic Machine Checks

On a real x64 system, a processor may have the ability to detect and report hardware (machine) errors. These are signaled by the processor by generating a machine-check exception (#MC). Some processors may also include a means to signal system or application level software to respond to certain uncorrected machine check errors in order to allow software to participate in the containment of and recovery from these errors.

The hypervisor provides a facility to inject a synthetic machine check on a virtual processor. This enables the operating system and application software running in a guest partition to be notified about physical platform errors, and to participate in any supported software error recovery scheme.

7.8 Virtual Processor Data Types

7.8.1 Virtual Processor Index

Virtual processors are identified by using an index (VP index). The maximum number of virtual processors per partition supported by the current implementation of the hypervisor can be obtained through CPUID leaf 0x40000005. A virtual processor index must be less than the maximum number of virtual processors per partition.

A special value HV_ANY_VP can be used in certain situations to specify “any virtual processor”.

```
typedef UINT32 HV_VP_INDEX;

#define HV_ANY_VP       ((HV_VP_INDEX)-1)

#define HV_VP_INDEX_SELF ((HV_VP_INDEX)-2)
```

A virtual processor’s ID can be retrieved by the guest through a hypervisor-defined MSR (model-specific register) HV_X64_MSR_VP_INDEX. A value of HV_VP_INDEX_SELF can be used to specify one’s own VP index.

```
#define HV_X64_MSR_VP_INDEX 0x40000002
```

7.8.2 Virtual Processor Register Names

Virtual processor state is referenced by *register names*, 32-bit numbers that uniquely identify a register.

```
typedef enum
{
   // Suspend Registers
   HvRegisterExplicitSuspend            = 0x00000000,
   HvRegisterInterceptSuspend           = 0x00000001,
   // Version
   // 128-bit result same as CPUID 0x40000002
   HvRegisterHypervisorVersion          = 0x00000100,

   // Feature Access (registers are 128 bits)
   // 128-bit result same as CPUID 0x40000003
   HvRegisterPrivilegesAndFeaturesInfo  = 0x00000200,

   // 128-bit result same as CPUID 0x40000004
   HvRegisterFeaturesInfo               = 0x00000201,

   // 128-bit result same as CPUID 0x40000005
   HvRegisterImplementationLimitsInfo   = 0x00000202,

   // 128-bit result same as CPUID 0x40000006
   HvRegisterHardwareFeaturesInfo       = 0x00000203,

   // Guest Crash Registers
   HvRegisterGuestCrashP0  = 0x00000210,
   HvRegisterGuestCrashP1  = 0x00000211,
   HvRegisterGuestCrashP2  = 0x00000212,
   HvRegisterGuestCrashP3  = 0x00000213,
   HvRegisterGuestCrashP4  = 0x00000214,
   HvRegisterGuestCrashCtl = 0x00000215,

   // Power State Configuration
   HvRegisterPowerStateConfigC1    = 0x00000220,
   HvRegisterPowerStateTriggerC1   = 0x00000221,
   HvRegisterPowerStateConfigC2    = 0x00000222,
   HvRegisterPowerStateTriggerC2   = 0x00000223,
   HvRegisterPowerStateConfigC3    = 0x00000224,
   HvRegisterPowerStateTriggerC3   = 0x00000225,

   // Frequency Registers
   HvRegisterProcessorClockFrequency = 0x00000240,
   HvRegisterInterruptClockFrequency = 0x00000241,

   // Idle Register
   HvRegisterGuestIdle = 0x00000250,

   // Guest Debug
   HvRegisterDebugDeviceOptions   = 0x00000260,
   // Pending Interruption Register
   HvRegisterPendingInterruption = 0x00010002,

   // Interrupt State register
   HvRegisterInterruptState = 0x00010003,

   // Pending Event Register
   HvRegisterPendingEvent0          = 0x00010004,
   HvRegisterPendingEvent1          = 0x00010005,

   // User-Mode Registers
   HvX64RegisterRax     = 0x00020000,
   HvX64RegisterRcx     = 0x00020001,
   HvX64RegisterRdx     = 0x00020002,
   HvX64RegisterRbx     = 0x00020003,
   HvX64RegisterRsp     = 0x00020004,
   HvX64RegisterRbp     = 0x00020005,
   HvX64RegisterRsi     = 0x00020006,
   HvX64RegisterRdi     = 0x00020007,
   HvX64RegisterR8      = 0x00020008,
   HvX64RegisterR9      = 0x00020009,
   HvX64RegisterR10     = 0x0002000A,
   HvX64RegisterR11     = 0x0002000B,
   HvX64RegisterR12     = 0x0002000C,
   HvX64RegisterR13     = 0x0002000D,
   HvX64RegisterR14     = 0x0002000E,
   HvX64RegisterR15     = 0x0002000F,
   HvX64RegisterRip     = 0x00020010,
   HvX64RegisterRflags  = 0x00020011,

   // Floating Point and Vector Registers
   HvX64RegisterXmm0    = 0x00030000,
   HvX64RegisterXmm1    = 0x00030001,
   HvX64RegisterXmm2    = 0x00030002,
   HvX64RegisterXmm3    = 0x00030003,
   HvX64RegisterXmm4    = 0x00030004,
   HvX64RegisterXmm5    = 0x00030005,
   HvX64RegisterXmm6    = 0x00030006,
   HvX64RegisterXmm7    = 0x00030007,
   HvX64RegisterXmm8    = 0x00030008,
   HvX64RegisterXmm9    = 0x00030009,
   HvX64RegisterXmm10   = 0x0003000A,
   HvX64RegisterXmm11   = 0x0003000B,
   HvX64RegisterXmm12   = 0x0003000C,
   HvX64RegisterXmm13   = 0x0003000D,
   HvX64RegisterXmm14   = 0x0003000E,
   HvX64RegisterXmm15   = 0x0003000F,
   HvX64RegisterFpMmx0  = 0x00030010,
   HvX64RegisterFpMmx1  = 0x00030011,
   HvX64RegisterFpMmx2  = 0x00030012,
   HvX64RegisterFpMmx3  = 0x00030013,
   HvX64RegisterFpMmx4  = 0x00030014,
   HvX64RegisterFpMmx5  = 0x00030015,
   HvX64RegisterFpMmx6  = 0x00030016,
   HvX64RegisterFpMmx7  = 0x00030017,
   HvX64RegisterFpControlStatus     = 0x00030018,
   HvX64RegisterXmmControlStatus    = 0x00030019,

   // Control Registers
   HvX64RegisterCr0     = 0x00040000,
   HvX64RegisterCr2     = 0x00040001,
   HvX64RegisterCr3     = 0x00040002,
   HvX64RegisterCr4     = 0x00040003,
   HvX64RegisterCr8     = 0x00040004,
   HvX64RegisterXfem = 0x00040005,

   // X64 Intermediate Control Registers
   HvX64RegisterIntermediateCr0    = 0x00041000,
   HvX64RegisterIntermediateCr4    = 0x00041003,
   HvX64RegisterIntermediateCr8    = 0x00041004,

   // Debug Registers
   HvX64RegisterDr0     = 0x00050000,
   HvX64RegisterDr1     = 0x00050001,
   HvX64RegisterDr2     = 0x00050002,
   HvX64RegisterDr3     = 0x00050003,
   HvX64RegisterDr6     = 0x00050004,
   HvX64RegisterDr7     = 0x00050005,

   // Segment Registers
   HvX64RegisterEs      = 0x00060000,
   HvX64RegisterCs      = 0x00060001,
   HvX64RegisterSs      = 0x00060002,
   HvX64RegisterDs      = 0x00060003,
   HvX64RegisterFs      = 0x00060004,
   HvX64RegisterGs      = 0x00060005,
   HvX64RegisterLdtr    = 0x00060006,
   HvX64RegisterTr      = 0x00060007,

   // Table Registers
   HvX64RegisterIdtr    = 0x00070000,
   HvX64RegisterGdtr    = 0x00070001,

   // Virtualized MSRs
   HvX64RegisterTsc            = 0x00080000,
   HvX64RegisterEfer           = 0x00080001,
   HvX64RegisterKernelGsBase   = 0x00080002,
   HvX64RegisterApicBase       = 0x00080003,
   HvX64RegisterPat            = 0x00080004,
   HvX64RegisterSysenterCs     = 0x00080005,
   HvX64RegisterSysenterRip    = 0x00080006,
   HvX64RegisterSysenterRsp    = 0x00080007,
   HvX64RegisterStar           = 0x00080008,
   HvX64RegisterLstar          = 0x00080009,
   HvX64RegisterCstar          = 0x0008000A,
   HvX64RegisterSfmask         = 0x0008000B,
   HvX64RegisterInitialApicId  = 0x0008000C,

   // Cache control MSRs
   HvX64RegisterMtrrCap        = 0x0008000D,
   HvX64RegisterMtrrDefType    = 0x0008000E,

   HvX64RegisterMtrrPhysBase0 = 0x00080010,
   HvX64RegisterMtrrPhysBase1 = 0x00080011,
   HvX64RegisterMtrrPhysBase2 = 0x00080012,
   HvX64RegisterMtrrPhysBase3 = 0x00080013,
   HvX64RegisterMtrrPhysBase4 = 0x00080014,
   HvX64RegisterMtrrPhysBase5 = 0x00080015,
   HvX64RegisterMtrrPhysBase6 = 0x00080016,
   HvX64RegisterMtrrPhysBase7 = 0x00080017,
   HvX64RegisterMtrrPhysBase8 = 0x00080018,
   HvX64RegisterMtrrPhysBase9 = 0x00080019,
   HvX64RegisterMtrrPhysBaseA = 0x0008001A,
   HvX64RegisterMtrrPhysBaseB = 0x0008001B,
   HvX64RegisterMtrrPhysBaseC = 0x0008001C,
   HvX64RegisterMtrrPhysBaseD = 0x0008001D,
   HvX64RegisterMtrrPhysBaseE = 0x0008001E,
   HvX64RegisterMtrrPhysBaseF = 0x0008001F,

   HvX64RegisterMtrrPhysMask0 = 0x00080040,
   HvX64RegisterMtrrPhysMask1 = 0x00080041,
   HvX64RegisterMtrrPhysMask2 = 0x00080042,
   HvX64RegisterMtrrPhysMask3 = 0x00080043,
   HvX64RegisterMtrrPhysMask4 = 0x00080044,
   HvX64RegisterMtrrPhysMask5 = 0x00080045,
   HvX64RegisterMtrrPhysMask6 = 0x00080046,
   HvX64RegisterMtrrPhysMask7 = 0x00080047,
   HvX64RegisterMtrrPhysMask8 = 0x00080048,
   HvX64RegisterMtrrPhysMask9 = 0x00080049,
   HvX64RegisterMtrrPhysMaskA = 0x0008004A,
   HvX64RegisterMtrrPhysMaskB = 0x0008004B,
   HvX64RegisterMtrrPhysMaskC = 0x0008004C,
   HvX64RegisterMtrrPhysMaskD = 0x0008004D,
   HvX64RegisterMtrrPhysMaskE = 0x0008004E,
   HvX64RegisterMtrrPhysMaskF = 0x0008004F,

   HvX64RegisterMtrrFix64k00000     = 0x00080070,
   HvX64RegisterMtrrFix16k80000     = 0x00080071,
   HvX64RegisterMtrrFix16kA0000     = 0x00080072,
   HvX64RegisterMtrrFix4kC0000      = 0x00080073,
   HvX64RegisterMtrrFix4kC8000      = 0x00080074,
   HvX64RegisterMtrrFix4kD0000      = 0x00080075,
   HvX64RegisterMtrrFix4kD8000      = 0x00080076,
   HvX64RegisterMtrrFix4kE0000      = 0x00080077,
   HvX64RegisterMtrrFix4kE8000      = 0x00080078,
   HvX64RegisterMtrrFix4kF0000      = 0x00080079,
   HvX64RegisterMtrrFix4kF8000      = 0x0008007A,

   HvX64RegisterBndcfgs             = 0x0008007C,
   HvX64RegisterDebugCtl            = 0x0008007D,


   HvX64RegisterSgxLaunchControl0  = 0x00080080,
   HvX64RegisterSgxLaunchControl1  = 0x00080081,
   HvX64RegisterSgxLaunchControl2  = 0x00080082,
   HvX64RegisterSgxLaunchControl3  = 0x00080083,

   // Other MSRs
   HvX64RegisterMsrIa32MiscEnable  = 0x000800A0,
   HvX64RegisterIa32FeatureControl = 0x000800A1,

   // Performance monitoring MSRs
   HvX64RegisterPerfGlobalCtrl     = 0x00081000,
   HvX64RegisterPerfGlobalStatus   = 0x00081001,
   HvX64RegisterPerfGlobalInUse    = 0x00081002,
   HvX64RegisterFixedCtrCtrl       = 0x00081003,
   HvX64RegisterDsArea             = 0x00081004,
   HvX64RegisterPebsEnable         = 0x00081005,
   HvX64RegisterPebsLdLat          = 0x00081006,
   HvX64RegisterPebsFrontend       = 0x00081007,
   HvX64RegisterPerfEvtSel0        = 0x00081100,
   HvX64RegisterPmc0               = 0x00081200,
   HvX64RegisterFixedCtr0          = 0x00081300,

   HvX64RegisterLbrTos             = 0x00082000,
   HvX64RegisterLbrSelect          = 0x00082001,
   HvX64RegisterLerFromLip         = 0x00082002,
   HvX64RegisterLerToLip           = 0x00082003,
   HvX64RegisterLbrFrom0           = 0x00082100,
   HvX64RegisterLbrTo0             = 0x00082200,
   HvX64RegisterLbrInfo0           = 0x00083300,

   // Hypervisor-defined MSRs (Misc)
   HvX64RegisterVpRuntime           = 0x00090000,
   HvX64RegisterHypercall           = 0x00090001,
   HvRegisterGuestOsId              = 0x00090002,
   HvRegisterVpIndex                = 0x00090003,
   HvRegisterTimeRefCount           = 0x00090004,
   HvRegisterCpuManagementVersion   = 0x00090007,

   // Virtual APIC registers MSRs
   HvX64RegisterEoi                 = 0x00090010,
   HvX64RegisterIcr                 = 0x00090011,
   HvX64RegisterTpr                 = 0x00090012,

   HvRegisterVpAssistPage           = 0x00090013,
   HvRegisterReferenceTsc           = 0x00090017,
   // Performance statistics MSRs
   HvRegisterStatsPartitionRetail   = 0x00090020,
   HvRegisterStatsPartitionInternal = 0x00090021,
   HvRegisterStatsVpRetail          = 0x00090022,
   HvRegisterStatsVpInternal        = 0x00090023,

   // Partition Timer Assist Registers
   HvX64RegisterEmulatedTimerPeriod    = 0x00090030,
   HvX64RegisterEmulatedTimerControl   = 0x00090031,
   HvX64RegisterPmTimerAssist          = 0x00090032,

   // Hypervisor-defined MSRs (Synic)
   HvRegisterSint0      = 0x000A0000,
   HvRegisterSint1      = 0x000A0001,
   HvRegisterSint2      = 0x000A0002,
   HvRegisterSint3      = 0x000A0003,
   HvRegisterSint4      = 0x000A0004,
   HvRegisterSint5      = 0x000A0005,
   HvRegisterSint6      = 0x000A0006,
   HvRegisterSint7      = 0x000A0007,
   HvRegisterSint8      = 0x000A0008,
   HvRegisterSint9      = 0x000A0009,
   HvRegisterSint10     = 0x000A000A,
   HvRegisterSint11     = 0x000A000B,
   HvRegisterSint12     = 0x000A000C,
   HvRegisterSint13     = 0x000A000D,
   HvRegisterSint14     = 0x000A000E,
   HvRegisterSint15     = 0x000A000F,
   HvRegisterScontrol   = 0x000A0010,
   HvRegisterSversion   = 0x000A0011,
   HvRegisterSifp       = 0x000A0012,
   HvRegisterSipp       = 0x000A0013,
   HvRegisterEom        = 0x000A0014,
   HvRegisterSirbp      = 0x000A0015,


   // Hypervisor-defined MSRs (Synthetic Timers)
   HvRegisterStimer0Config            = 0x000B0000,
   HvRegisterStimer0Count             = 0x000B0001,
   HvRegisterStimer1Config            = 0x000B0002,
   HvRegisterStimer1Count             = 0x000B0003,
   HvRegisterStimer2Config            = 0x000B0004,
   HvRegisterStimer2Count             = 0x000B0005,
   HvRegisterStimer3Config            = 0x000B0006,
   HvRegisterStimer3Count             = 0x000B0007,
   HvRegisterStimeUnhaltedTimerConfig = 0x000B0100,
   HvRegisterStimeUnhaltedTimerCount  = 0x000B0101,

    //
    // XSAVE/XRSTOR register names.
    //

    // XSAVE AFX extended state registers.
    HvX64RegisterYmm0Low             = 0x000C0000,
    HvX64RegisterYmm1Low             = 0x000C0001,
    HvX64RegisterYmm2Low             = 0x000C0002,
    HvX64RegisterYmm3Low             = 0x000C0003,
    HvX64RegisterYmm4Low             = 0x000C0004,
    HvX64RegisterYmm5Low             = 0x000C0005,
    HvX64RegisterYmm6Low             = 0x000C0006,
    HvX64RegisterYmm7Low             = 0x000C0007,
    HvX64RegisterYmm8Low             = 0x000C0008,
    HvX64RegisterYmm9Low             = 0x000C0009,
    HvX64RegisterYmm10Low            = 0x000C000A,
    HvX64RegisterYmm11Low            = 0x000C000B,
    HvX64RegisterYmm12Low            = 0x000C000C,
    HvX64RegisterYmm13Low            = 0x000C000D,
    HvX64RegisterYmm14Low            = 0x000C000E,
    HvX64RegisterYmm15Low            = 0x000C000F,
    HvX64RegisterYmm0High            = 0x000C0010,
    HvX64RegisterYmm1High            = 0x000C0011,
    HvX64RegisterYmm2High            = 0x000C0012,
    HvX64RegisterYmm3High            = 0x000C0013,
    HvX64RegisterYmm4High            = 0x000C0014,
    HvX64RegisterYmm5High            = 0x000C0015,
    HvX64RegisterYmm6High            = 0x000C0016,
    HvX64RegisterYmm7High            = 0x000C0017,
    HvX64RegisterYmm8High            = 0x000C0018,
    HvX64RegisterYmm9High            = 0x000C0019,
    HvX64RegisterYmm10High           = 0x000C001A,
    HvX64RegisterYmm11High           = 0x000C001B,
    HvX64RegisterYmm12High           = 0x000C001C,
    HvX64RegisterYmm13High           = 0x000C001D,
    HvX64RegisterYmm14High           = 0x000C001E,
    HvX64RegisterYmm15High           = 0x000C001F


    // Synthetic VSM registers
    //


    HvRegisterVsmCodePageOffsets     = 0x000D0002,
    HvRegisterVsmVpStatus            = 0x000D0003,
    HvRegisterVsmPartitionStatus     = 0x000D0004,
    HvRegisterVsmVina                = 0x000D0005,
    HvRegisterVsmCapabilities        = 0x000D0006,
    HvRegisterVsmPartitionConfig     = 0x000D0007,

    HvRegisterVsmVpSecureConfigVtl0  = 0x000D0010,
    HvRegisterVsmVpSecureConfigVtl1  = 0x000D0011,
    HvRegisterVsmVpSecureConfigVtl2  = 0x000D0012,
    HvRegisterVsmVpSecureConfigVtl3  = 0x000D0013,
    HvRegisterVsmVpSecureConfigVtl4  = 0x000D0014,
    HvRegisterVsmVpSecureConfigVtl5  = 0x000D0015,
    HvRegisterVsmVpSecureConfigVtl6  = 0x000D0016,
    HvRegisterVsmVpSecureConfigVtl7  = 0x000D0017,
    HvRegisterVsmVpSecureConfigVtl8  = 0x000D0018,
    HvRegisterVsmVpSecureConfigVtl9  = 0x000D0019,
    HvRegisterVsmVpSecureConfigVtl10 = 0x000D001A,
    HvRegisterVsmVpSecureConfigVtl11 = 0x000D001B,
    HvRegisterVsmVpSecureConfigVtl12 = 0x000D001C,
    HvRegisterVsmVpSecureConfigVtl13 = 0x000D001D,
    HvRegisterVsmVpSecureConfigVtl14 = 0x000D001E,

    HvRegisterVsmVpWaitForTlbLock    = 0x000D0020,
} HV_REGISTER_NAME;
```

7.8.3 Virtual Processor Register Values

Virtual processor register values are all 128 bits in size. Values that do not consume the full 128 bits are zero-extended to fill out the entire 128 bits.

```
typedef union
{
   UINT128                                  Reg128;
   UINT64                                   Reg64;
   UINT32                                   Reg32;
   UINT16                                   Reg16;
   UINT8                                    Reg8;
   HV_X64_FP_REGISTER                       Fp;
   HV_X64_FP_CONTROL_STATUS_REGISTER        FpControlStatus;
   HV_X64_XMM_CONTROL_STATUS_REGISTER       XmmControlStatus;
   HV_X64_SEGMENT_REGISTER                  Segment;
   HV_X64_TABLE_REGISTER                    Table;
   HV_EXPLICIT_SUSPEND_REGISTER             ExplicitSuspend;
   HV_INTERCEPT_SUSPEND_REGISTER            InterceptSuspend;
   HV_X64_INTERRUPT_STATE_REGISTER          InterruptState;
   HV_X64_PENDING_INTERRUPTION_REGISTER     PendingInterruption;
   HV_X64_MSR_NPIEP_CONFIG_CONTENTS         NpiepConfig;
   HV_X64_PENDING_EXCEPTION_EVENT           PendingExceptionEvent;
} HV_REGISTER_VALUE;

typedef HV_REGISTER_VALUE *PHV_REGISTER_VALUE;
```

7.8.4 Synthetic Machine Check Status Data Structure

```
typedef union _HV_X64_MSR_SYNMC_STATUS_CONTENTS
{
    struct
    {
        UINT16 McaErrorCode;

        union
        {
            UINT16 ModelSpecificErrorCode;

            struct
            {
                UINT16 ErrorDetail      : 14;
                UINT16 HypervisorError  : 1;
                UINT16 SoftwareError    : 1;
            };
        };

        struct
        {
            UINT32 Reserved             : 23;
            UINT32 ActionRequired       : 1;
            UINT32 Signaling            : 1;
            UINT32 ContextCorrupt       : 1;  // Hypervisor/virt stack
                                              // context corrupt
            UINT32 AddressValid         : 1;
            UINT32 MiscValid            : 1;
            UINT32 ErrorEnabled         : 1;
            UINT32 Uncorrected          : 1;  // Uncorrected error
            UINT32 Overflow             : 1;  // Error overflow
            UINT32 Valid                : 1;  // Register valid
        };
    };

    UINT64 AsUINT64;

} HV_X64_MSR_SYNMC_STATUS_CONTENTS;
```

7.8.5 Synthetic Machine Check Error Code

```
#define HV_SYNMC_MCA_ERROR_CODE (0x0001)        // Unclassified error
```

7.8.6 Synthetic Machine Check Event Data Structure

```
typedef struct _HV_SYNMC_X64_EVENT
{
    HV_X64_MSR_SYNMC_STATUS_CONTENTS Status;
    HV_X64_MSR_SYNMC_ADDR_CONTENTS   Addr;
    HV_X64_MSR_SYNMC_MISC_CONTENTS   Misc;
    BOOLEAN                          RipValid;
    BOOLEAN                          EipValid;

} HV_SYNMC_X64_EVENT;
```

7.8.7 Virtual Processor Assist Page

The hypervisor provides a page per virtual processor which is overlaid on the guest GPA space. This page can be used for bi-directional communication between a guest VP and the hypervisor. The guest OS has read/write access to this virtual VP assist page.

7.8.7.1 VP Assist Page Register

A guest specifies the location of the overlay page (in GPA space) by writing to the Virtual VP Assist MSR (0x40000073). The format of the Virtual VP Assist Page MSR is as follows:

63:12	11:1	0
Virtual VP Assist Page PFN	RsvdP	Enable

7.8.7.2 **VP Assist Page Format**

```
typedef union _HV_VP_ASSIST_PAGE
{
    struct
    {
        //
        // APIC assist for optimized EOI processing.
        //

        HV_VIRTUAL_APIC_ASSIST ApicAssist;

        UINT32 ReservedZ0;

        //
        // VP-VTL control information
        //

        HV_VP_VTL_CONTROL VtlControl;

        HV_NESTED_ENLIGHTENMENTS_CONTROL NestedEnlightenmentsControl;

        BOOLEAN EnlightenVmEntry;

        UINT8 ReservedZ1[7];

        HV_GPA CurrentNestedVmcs;

        BOOLEAN SyntheticTimeUnhaltedTimerExpired;

        UINT8 ReservedZ2[7];

        //
        // VirtualizationFaultInformation must be 16 byte aligned.
        //

        HV_VIRTUALIZATION_FAULT_INFORMATION VirtualizationFaultInformation;

    };

    UINT8 ReservedZBytePadding[HV_PAGE_SIZE];

} HV_VP_ASSIST_PAGE, *PHV_VP_ASSIST_PAGE;
```

The format of the structures used within the VP Assist Page are described in the following sections.

Nested Enlightenments Control

```
//
// Control structure that allows a hypervisor to indicate to its parent
// hypervisor which nested enlightenment privileges are to be granted to the
// current nested guest context.
//

typedef struct _HV_NESTED_ENLIGHTENMENTS_CONTROL
{
    union
    {
        UINT32 AsUINT32;

        struct
        {
            UINT32 DirectHypercall:1;
            UINT32 VirtualizationException:1;
            UINT32 Reserved:30;
        };

    } Features;

    union
    {
        UINT32 AsUINT32;

        struct
        {
            UINT32 InterPartitionCommunication:1;
            UINT32 Reserved:31;
        };

    } HypercallControls;

} HV_NESTED_ENLIGHTENMENTS_CONTROL, *PHV_NESTED_ENLIGHTENMENTS_CONTROL;
```

Virtualization Fault Information Area

```
//
// The virtualization fault information area contains the current fault code
and
// fault parameters for the VP. It is 16 byte aligned.
//

typedef union _HV_VIRTUALIZATION_FAULT_INFORMATION
{
    struct
    {
        UINT16 Parameter0;
        UINT16 Reserved0;
        UINT32 Code;
        UINT64 Parameter1;
    };

} HV_VIRTUALIZATION_FAULT_INFORMATION, *PHV_VIRTUALIZATION_FAULT_INFORMATION;
```

7.8.7.3 Virtual Processor Set

A *virtual processor set* represents a collection of virtual processors, and can be used as an input for some hypercalls.

```
typedef struct
{
    UINT64 Format;
    UINT64 ValidBanksMask;
    UINT64 BankContents[];
} HV_VP_SET
```

A processor set has two modes, which are specified by the format field. Processor sets with a format "1" represent all virtual processors for the given partition. Processor sets with a format "0" describe a sparse set of virtual processors.

```
typedef enum {
    HvGenericSetSparse4k,
    HvGenericSetAll,
} HV_GENERIC_SET_FORMAT
```

Set behavior	"format" value
A sparse subset of VPs	0
All VPs (belonging to a partition)	1

7.8.7.4 Sparse Virtual Processor Set

The following section describes how to construct a sparse set of virtual processors.

The total set of virtual processors is split up into chunks of 64, known as a "bank". For example, processors 0-63 are in bank 0, 64-127 are in bank 1, and so on.

To describe an individual processor, its bank is specified with ValidBanksMask. Each bit in ValidBanksMask represents a particular bank.

$$bank = \frac{VP\ index}{64}$$

For every bit that is set with ValidBanksMask, there must be an element in the BanksContents array. This element is a mask describing the bank itself.

If a bit in ValidBankMask is 0, there is no corresponding element in BanksContents. Furthermore, for a bit 1 in ValidBankMask, it is valid state for the corresponding element in BanksContents can be all 0s, meaning no processors are specified in this bank.

7.8.7.5 Processor Set Example

Suppose a partition has 200 VPs, and we wish to specify the following set:

$$\{\,0, 5, 130\,\}$$

First, the format is 0, since this is a sparse set.

Next, the corresponding banks (and therefore the set bits of ValidBanksMask) are

$$\{\,0, 0, 2\,\}$$

Thus, ValidBanksMask is 0x05.

Bank 0 sets bits 0 and 5 to specify the VPs within that bank. Therefore, the corresponding element in the BankContents mask is 0x21.

Since bit 1 is not set in ValidBanksMask, there is no corresponding element in BankContents.

Bank 2 represents VP indices 128-191. To describe index 130, bit 2 of the corresponding mask is set. Thus, BankContents is:

$$\{ 0x21, 0x04 \}$$

7.9 Virtual Processor Register Formats

7.9.1 Virtual Processor Run Time Register

The hypervisor's scheduler internally tracks how much time each virtual processor consumes in executing code. The time tracked is a combination of the time the virtual processor consumes running guest code, and the time the associated logical processor spends running hypervisor code on behalf of that guest. This cumulative time is accessible through the 64-bit read-only HV_X64_MSR_VP_RUNTIME hypervisor MSR. The time quantity is measured in 100ns units.

63:0
VP Runtime

7.9.2 Virtual Processor Interrupt State Register

The *interrupt state* register provides information about the interrupt state of the virtual processor. It indicates whether the virtual processor is in an "interrupt shadow" and whether non-maskable interrupts are currently masked. Certain instructions inhibit the delivery of hardware interrupts and debug traps for one instruction. Furthermore, when a non-maskable interrupt is delivered to the virtual processor, subsequent non-maskable interrupts are masked until the virtual processor executes an IRET instruction.

```
The interrupt state register is encoded as follows:
typedef struct
{
   UINT64   InterruptShadow:1;
   UINT64   NmiMasked:1;
   UINT64   Reserved:62;
} HV_X64_INTERRUPT_STATE_REGISTER;
```

7.9.3 Virtual Processor Pending Interruption Register

The *pending interruption* register is used to indicate whether a pending interruption exists for the virtual processor. An interruption is defined as any event that is delivered through the virtual processor's interrupt descriptor table (for example, exceptions, interrupts, or debug traps). If an interruption is pending, the hypervisor will generate the interruption when the virtual processor resumes execution. This allows code running within the parent partition, for example, to respond to an MSR intercept by generating a general protection fault.

If an intercept is generated during the delivery of an interruption, the interruption is held pending and an intercept message is sent to the parent partition. The parent partition can resolve the intercept and resume the virtual processor, in which case the interruption will be re-delivered.

The type of a pending interruption is encoded as follows:

```
typedef enum
{
   HvX64PendingInterrupt                     = 0,
   HvX64PendingNmi                           = 2,
   HvX64PendingException                     = 3,
   HvX64PendingSoftwareInterrupt             = 4,
   HvX64PendingPrivilegedSoftwareException = 5,
   HvX64PendingSoftwareException             = 6
} HV_X64_PENDING_INTERRUPTION_TYPE;

The format of the pending interruption register is as follows:
typedef struct
{
   UINT32   InterruptionPending:1;
   UINT32   InterruptionType:3
   UINT32   DeliverErrorCode:1;
   UINT32   InstructionLength:4;
   UINT32   Reserved:7;
   UINT32   InterruptionVector:16;
   UINT32   ErrorCode;
} HV_X64_PENDING_INTERRUPTION_REGISTER;
```

If the *InterruptionPending* bit is cleared, no interruption is pending, and the values in the other fields are ignored.

InterruptionType indicates the type of the interruption and can be any of the following values:

- *HVX64PendingInterrupt* — The interruption is due to an interrupt.
- *HVX64PendingNmi* — The interruption is due to a non-maskable interrupt.
- *HVX64PendingException* — The interruption is due to a hardware exception.
- *HVX64PendingSoftwareInterrupt* – The interruption is due to a software interrupt.
- *HVX64PendingPrivilegedSoftwareException* – The interruption is due to a software exception from privileged software, such as a debug trap or fault.
- *HVX64PendingSoftwareException* – The interruption is due to a software exception.

If the interruption is of type HvX64PendingSoftwareIntertupt, the *InstructionLength* field indicates the length of the instruction in bytes.

DeliverErrorCode indicates whether an error code should be pushed on the stack as part of the interruption.

InterruptionVector indicates the vector to use for the exception.

ErrorCode indicates the error code value that will be pushed as part of the interruption frame.

7.9.4 **Virtual Processor Floating-point and Vector Registers**

Floating point registers are encoded as 80-bit values, as follows:

```
typedef struct
{
   UINT64   Mantissa;
   UINT64   BiasedExponent:15;
   UINT64   Sign:1;
   UINT64   Reserved:48;
} HV_X64_FP_REGISTER;
```

Additional status and control information for the floating point and vector units are stored in the following formats:

```
typedef struct
{
   UINT16   FpControl;
   UINT16   FpStatus;
   UINT8    FpTag;
   UINT8    Reserved:8;
   UINT16   LastFpOp;
   union
   {
      UINT64      LastFpRip;
      struct
      {
            UINT32      LastFpEip;
            UINT16      LastFpCs;
      };
   };
} HV_X64_FP_CONTROL_STATUS_REGISTER;

typedef struct
{
   union
   {
      UINT64      LastFpRdp;
      struct
      {
            UINT32      LastFpDp;
            UINT16      LastFpDs;
      };
   };
   UINT32   XmmStatusControl;
   UINT32   XmmStatusControlMask;
} HV_X64_XMM_CONTROL_STATUS_REGISTER;
```

7.9.5 **Virtual Processor Segment Registers**

Segment register state is encoded as follows:

```
typedef struct
{
   UINT64   Base;
   UINT32   Limit;
   UINT16   Selector;
   union
   {
      struct
      {
            UINT16      SegmentType:4;
            UINT16      NonSystemSegment:1;
            UINT16      DescriptorPrivilegeLevel:2;
            UINT16      Present:1;
            UINT16      Reserved:4;
            UINT16      Available:1;
            UINT16      Long:1;
            UINT16      Default:1;
            UINT16      Granularity:1;
      };

      UINT16      Attributes;
   };
} HV_X64_SEGMENT_REGISTER;
```

The limit is encoded as a 32-bit value. For X64 long-mode segments, the limit is ignored. For legacy x86 segments, the limit must be expressible within the bounds of the x64 processor architecture. For example, if the "G" (granularity) bit is set within the attributes of a code or data segment, the low-order 12 bits of the limit must be 1s.

The "Present" bit controls whether the segment acts like a null segment (that is, whether a memory access performed through that segment generates a #GP fault).

The MSRs IA32_FS_BASE and IA32_GS_BASE are not defined in the register list, as they are aliases to the base element of the segment register structure. Use HvX64RegisterFs and HvX64RegisterGs and the structure above instead.

7.9.6 **Virtual Processor Table Registers**

Table registers are similar to segment registers, but they have no selector or attributes, and the limit is restricted to 16 bits.

```
typedef struct
{
   UINT16   Pad[3];
   UINT16   Limit;
   UINT64   Base;
} HV_X64_TABLE_REGISTER;
```

7.9.7 **Synthetic Machine Check Registers**

```
typedef UINT64 HV_X64_MSR_SYNMC_ADDR_CONTENTS

typedef UINT64 HV_X64_MSR_SYNMC_MISC_CONTENTS
```

7.10 Virtual Processor Interfaces

7.10.1 **HvSetVpRegisters**

The HvSetVpRegisters hypercall writes the architectural state of a virtual processor.

Wrapper Interface

```
HV_STATUS
HvSetVpRegisters(
   __in    HV_PARTITION_ID    PartitionId,
   __in    HV_VP_INDEX  VpIndex,
   __in       HV_INPUT_VTL    InputVtl,
   __inout PUINT32      RegisterCount,
   __in_ecount(RegisterCount)
        PCHV_REGISTER_NAME    RegisterNameList,
   __in_ecount(RegisterCount)
        PCHV_REGISTER_VALUE   RegisterValueList
   );
```

Native Interface

HvSetVpRegisters [rep]			
	Call Code = 0x0051		
🡆 Input Parameter Header			
0	PartitionId (8 bytes)		
8	VpIndex (4 bytes)	TargetVtl (1 byte)	RsvdZ (3 bytes)
🡆 Input List Element			
0	RegisterName (4 bytes)	Padding (4 bytes)	
16	RegisterValue (low-order) (8 bytes)		
24	RegisterValue (high-order) (8 bytes)		

Description

The state is written as a series of register values, each corresponding to a register name provided as input.

Minimal error checking is performed when a register value is modified. In particular, the hypervisor will validate that reserved bits of a register are set to zero, bits that are architecturally defined as always containing a zero or a one are set appropriately, and specified bits beyond the architectural size of the register are zeroed.

This call cannot be used to modify the value of a read-only register.

Side-effects of modifying a register are not performed. This includes generation of exceptions, pipeline synchronizations, TLB flushes, and so on.

Input Parameters

PartitionId specifies the partition.

VpIndex specifies the index of the virtual processor.

TargetVtl specifies the VTL to target.

RegisterName specifies the name of a register to be modified.

RegisterValue specifies the new value for the specified register.

Output Parameters

None.

Restrictions

- The partition specified by *PartitionId* must be in the “active” state.
- The caller must either be the parent of the partition specified by *PartitionId,* or the partition specified must be “self” and the partition must have the AccessVpRegisters privilege. See 4.2.2.
- Guest operating systems may assume that virtual processors are neither hot-added nor hot-removed from a partition during normal execution. See 7.10.3.
- This hypercall is a supported way to set the following registers. All other registers should not be set using this hypercall.

Register
HvRegisterPendingEvent0 = 0x00010004,
HvRegisterVpAssistPage = 0x00090013,
HvRegisterVsmCodePageOffsets = 0x000D0002,
HvRegisterVsmVina = 0x000D0005,
HvRegisterVsmPartitionConfig = 0x000D0007,
HvRegisterVsmVpSecureConfigVtl0 = 0x000D0010,
HvX64RegisterRsp = 0x00020004,
HvX64RegisterRip = 0x00020010,
HvX64RegisterRflags = 0x00020011,
HvX64RegisterCr3 = 0x00040002,
HvX64RegisterCr8 = 0x00040004,
HvX64RegisterDr7 = 0x00050005,
HvX64RegisterEs = 0x00060000,
HvX64RegisterCs = 0x00060001,
HvX64RegisterSs = 0x00060002,
HvX64RegisterDs = 0x00060003,
HvX64RegisterFs = 0x00060004,
HvX64RegisterGs = 0x00060005,
HvX64RegisterLdtr = 0x00060006,
HvX64RegisterTr = 0x00060007,
HvX64RegisterIdtr = 0x00070000,
HvX64RegisterGdtr = 0x00070001,
HvX64RegisterEfer = 0x00080001,
HvX64RegisterKernelGsBase = 0x00080002,
HvX64RegisterSysenterCs = 0x00080005,

Register
HvX64RegisterStar = 0x00080008,
HvX64RegisterLstar = 0x00080009,
HvX64RegisterCstar = 0x0008000A,
HvX64RegisterSfmask = 0x0008000B,
HvX64RegisterApicBase = 0x00080003,
HvX64RegisterCrInterceptControl = 0x000E0000,
HvX64RegisterCrInterceptCr0Mask = 0x000E0001,
HvX64RegisterCrInterceptCr4Mask = 0x000E0002,
HvX64RegisterCrInterceptIa32MiscEnableMask = 0x000E0003,

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition is invalid.
HV_STATUS_INVALID_VP_INDEX	The specified VP index does not reference a virtual processor within the specified partition.
HV_STATUS_INVALID_VP_STATE	The state of the specified VP does not allow the requested operation.
HV_STATUS_INVALID_PARAMETER	The specified register name is invalid.
	The specified register is read-only.
	The specified register value is not valid (for example, a reserved bit is not zero).
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INVALID_REGISTER_VALUE	The supplied register value is invalid.

7.10.2 **HvGetVpRegisters**

The HvGetVpRegisters hypercall reads the architectural state of a virtual processor.

Wrapper Interface

```
HV_STATUS
HvGetVpRegisters(
   __in    HV_PARTITION_ID    PartitionId,
   __in    HV_VP_INDEX  VpIndex,
   __in       HV_INPUT_VTL    InputVtl,
   __inout PUINT32      RegisterCount,
   __in_ecount(RegisterCount)
        PCHV_REGISTER_NAME    RegisterNameList,
   __out_ecount(RegisterCount)
        PHV_REGISTER_VALUE    RegisterValueList
   );
```

Native Interface

HvGetVpRegisters [rep]			
	Call Code = 0x0050		
➡ Input Parameter Header			
0	PartitionId (8 bytes)		
8	VpIndex (4 bytes)	InputVtl (1 byte)	Padding (3 bytes)
➡ Input List Element			
0	RegisterName[0] (4 bytes)	RegisterName[1] (4 bytes)	
⬅ Output List Element			
0	RegisterValue (low-order) (8 bytes)		
8	RegisterValue (high-order) (8 bytes)		

Description

The state is returned as a series of register values, each corresponding to a register name provided as input.

Input Parameters

PartitionId specifies the partition.

VpIndex specifies the index of the virtual processor.

TargetVtl specifies the VTL to target.

RegisterName specifies a list of names for the requested register state.

Output Parameters

RegisterValue returns a list of register values for the requested register state.

Restrictions

- The partition specified by *PartitionId* must be in the “active” state.
- The caller must be the parent of the partition specified by *PartitionId* or the partition specifying its own partition ID.
- Guest operating systems may assume that virtual processors are neither hot-added nor hot-removed from a partition during normal execution. See 7.10.3.
- This hypercall is a supported way to get the following registers. All other registers should not be read using this hypercall.

Register
HvRegisterPendingEvent0 = 0x00010004,
HvRegisterVpAssistPage = 0x00090013,
HvRegisterVsmCodePageOffsets = 0x000D0002,
HvRegisterVsmVina = 0x000D0005,
HvRegisterVsmPartitionConfig = 0x000D0007,
HvRegisterVsmVpSecureConfigVtl0 = 0x000D0010,
HvX64RegisterRsp = 0x00020004,
HvX64RegisterRip = 0x00020010,
HvX64RegisterRflags = 0x00020011,
HvX64RegisterCr3 = 0x00040002,
HvX64RegisterCr8 = 0x00040004,
HvX64RegisterDr7 = 0x00050005,
HvX64RegisterEs = 0x00060000,
HvX64RegisterCs = 0x00060001,
HvX64RegisterSs = 0x00060002,
HvX64RegisterDs = 0x00060003,
HvX64RegisterFs = 0x00060004,
HvX64RegisterGs = 0x00060005,
HvX64RegisterLdtr = 0x00060006,
HvX64RegisterTr = 0x00060007,
HvX64RegisterIdtr = 0x00070000,
HvX64RegisterGdtr = 0x00070001,
HvX64RegisterEfer = 0x00080001,
HvX64RegisterKernelGsBase = 0x00080002,
HvX64RegisterSysenterCs = 0x00080005,
HvX64RegisterStar = 0x00080008,
HvX64RegisterLstar = 0x00080009,
HvX64RegisterCstar = 0x0008000A,
HvX64RegisterSfmask = 0x0008000B,
HvX64RegisterApicBase = 0x00080003,
HvX64RegisterCrInterceptControl = 0x000E0000,
HvX64RegisterCrInterceptCr0Mask = 0x000E0001,
HvX64RegisterCrInterceptCr4Mask = 0x000E0002,

Register
HvX64RegisterCrInterceptIa32MiscEnableMask = 0x000E0003,

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is neither the partition itself nor the parent of the specified partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition is invalid.
HV_STATUS_INVALID_VP_INDEX	The specified VP index does not reference a virtual processor within the specified partition.
HV_STATUS_INVALID_PARAMETER	The specified register name is invalid.

7.10.3 **HvStartVirtualProcessor**

HvStartVirtualProcessor is an enlightened method for starting a virtual processor. It is functionally equivalent to traditional INIT-based methods, except that the VP can start with a desired register state.

Wrapper Interface

```
HV_STATUS
HVStartVirtualProcessor(
    __in HV_PARTITION_ID            PartitionId;
    __in HV_VP_INDEX                VpIndex;
    __in HV_VTL                     TargetVtl;
    __in HV_INITIAL_VP_CONTEXT      VpContext;
} HV_INPUT_START_VIRTUAL_PROCESSOR, *PHV_INPUT_START_VIRTUAL_PROCESSOR;
```

Input Structures

```
typedef struct
{
    UINT64 Rip;
    UINT64 Rsp;
    UINT64 Rflags;

    // Segment selector registers together with their hidden state.
    HV_X64_SEGMENT_REGISTER Cs;
    HV_X64_SEGMENT_REGISTER Ds;
    HV_X64_SEGMENT_REGISTER Es;
    HV_X64_SEGMENT_REGISTER Fs;
    HV_X64_SEGMENT_REGISTER Gs;
    HV_X64_SEGMENT_REGISTER Ss;
    HV_X64_SEGMENT_REGISTER Tr;
    HV_X64_SEGMENT_REGISTER Ldtr;

    // Global and Interrupt Descriptor tables
    HV_X64_TABLE_REGISTER Idtr;
    HV_X64_TABLE_REGISTER Gdtr;

    // Control registers and MSR's
    UINT64 Efer;
    UINT64 Cr0;
    UINT64 Cr3;
    UINT64 Cr4;
    UINT64 MsrCrPat;

} HV_INITIAL_VP_CONTEXT;
```

Native Interface

HvStartVirtualProcessor			
	Call Code = 0x0099		
➡ Input Parameters			
0	PartitionId (8 bytes)		
8	VpIndex (4 bytes)	TargetVtl (1 byte	Padding (3 bytes)
16	Rip (8 bytes)		
24	Rsp (8 bytes)		
32	Rflags (8 bytes)		

40	Cs[0] (8 bytes)
48	Cs[1] (8 bytes)
56	Ds[0] (8 bytes)
64	Ds[1] (8 bytes)
72	Es[0] (8 bytes)
80	Es[1] (8 bytes)
88	Fs[0] (8 bytes)
96	Fs[1] (8 bytes)
104	Gs[0] (8 bytes)
112	Gs[1] (8 bytes)
120	Ss[0] (8 bytes)
128	Ss[1] (8 bytes)
136	Ts[0] (8 bytes)
144	Ts[1] (8 bytes)
152	Ltdr[0] (8 bytes)
160	Ltdr[1] (8 bytes)
168	Itdr[0] (8 bytes)
176	Itdr[1] (8 bytes)
184	Gtdr[0] (8 bytes)
192	Gtdr[1] (8 bytes)
200	Efer (8 bytes)
208	Cr0 (8 bytes)
216	Cr3 (8 bytes)
224	Cr4 (8 bytes)
232	MsrCrPat (8 bytes)

Input Parameters

PartitionId specifies the partition.

VpIndex specifies the VP index to start. To get the VP index from an APIC ID, use HvGetVpIndexFromApicId.

TargetVtl specifies the target VTL

VpContext specifies the initial context in which this VP should start.

Description

HvStartVirtualProcessor is an enlightened method for starting a virtual processor. The VP will start with the specified control register state in protected/long mode, skipping real mode entirely.

This is the only method for starting a VP in a non-zero VTL.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Access denied.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_VP_INDEX	The virtual processor specified by HV_VP_INDEX is invalid.
HV_STATUS_INVALID_REGISTER_VALUE	The supplied register value is invalid.
HV_STATUS_INVALID_VP_STATE	A virtual processor is not in the correct state for the performance of the indicated operation.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INVALID_VTL_STATE	The VTL state conflicts with the requested operation.

7.10.4 **HvGetVpIndexFromApicId**

The HvGetVpIndexFromApicId is related to HvStartVirtualProcessor. It allows the caller to retrieve a VP index before the VP has started running.

Wrapper Interface

```
HV_STATUS
HvGetVpIndexFromApicId(
   __in HV_PARTITION_ID             PartitionId,
   __in HV_VTL                      TargetVtl,
   __inout PUINT32                  ApicIdCount,
   __in_ecount(ApicIdCount)
           PHV_APIC_ID  ApicIdList
   __out_ecount(RegisterCount)
        PHV_VP_INDEX    VpIndexList
   );
```

Native Interface

HvGetVpIndexFromApicId [rep]		
	Call Code = 0x009A	
➡ Input Parameter Header		
0	PartitionId (8 bytes)	
8	TargetVtl (1 byte)	Padding (7 bytes)
➡ Input List Element		
0	ApicId (4 bytes)	Padding (4 bytes)
⬅ Output List Element		
0	VpIndex (4 bytes)	Padding (4 bytes)

Input Parameters

PartitionId specifies the partition.

TargetVtl specifies the VTL of the target VP.

ApicId specifies the APIC ID of the VP.

Output Values

VpIndex is the VP with the APIC ID specified.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition, or the caller does not have the AccessVpIndex privilege. See section 4.2.2.
HV_STATUS_INVALID_PARTITION_ID	The specified partition is invalid.
HV_STATUS_INVALID_VP_INDEX	The specified VP index references an existing virtual processor within the specified partition.
	The specified VP index exceeds the maximum index allowed by the hypervisor implementation.
HV_STATUS_INVALID_PARAMETER	A parameter is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INVALID_VTL_STATE	The VTL specified is invalid.

8 Virtual Processor Execution

The virtual machine interface exposed by each partition exposes virtual processors, and these are architecture specific. This section specifies the core CPU aspects of virtual processors. The following two sections specify the MMU (memory management unit) and interrupt controller aspects of virtual processors.

A complete definition of virtual processor behavior requires hundreds of pages of CPU manuals. This document specifies the behavior of virtual processors by referencing processor manuals for physical X64 processors and discussing only cases where a virtual processor's behavior differs from that of a logical processor; that is, the baseline behavior of a virtual processor is defined by the Intel and AMD processor reference manuals.

8.1 Processor Features and CPUID

The processor intercept mechanism of the hypervisor allows a parent partition to intercept the execution of the CPUID instruction by virtual processors within its child partitions. The parent partition can set the return values of the CPUID instruction in arbitrary ways. Doing so does not automatically alter the set of processor features of a virtual processor; that is, if a parent partition chooses to alter the behavior of the CPUID instruction, it is responsible for ensuring that the set of virtual processor features matches what is indicated by the CPUID instruction.

8.2 Family, Model and Stepping Reported by CPUID

The CPUID instruction can be used to obtain a logical processor's family, model and stepping information. It is possible, but not guaranteed, that logical processors reporting differing information may coexist on a single system. To properly use this information, a partition must be able to execute with a hard affinity between virtual and logical processors. Only the root partition executes in this manner. As a result, the hypervisor will expose the true family, model and stepping only to the root partition and will report the minimum value detected in the logical processor configuration to all other partitions.

8.3 Platform ID Reported by MSR

The value returned by the IA32_PLATFORM_ID MSR (0x17) can be used in conjunction with the family, model and stepping information as reported by the CPUID instruction (as described in the previous section). The hypervisor is consistent in its handling of this MSR and will expose the true content of the IA32_PLATFORM_ID MSR only to the root partition. Guest access to this register is denied.

8.4 Real Mode

The hypervisor attempts to support real mode in a fully transparent manner. There may be situations, however, where specific processor implementations may not make this entirely possible. As a result, the hypervisor may be required to emulate or manipulate the environment to some degree to provide real mode support. The following is a list of potential areas where real mode support may not be transparent.

- Hypervisor overhead inconsistencies. As a consequence of an increase in the frequency of instruction emulation by the hypervisor, performance of both the guest and the system may be affected.

- Visible processor state changes as a consequence of switching modes. The hypervisor may be required to make changes to the guest's runtime environment when mode switches occur, such as between real and protected mode or vice versa. Such changes may be detected by the guest.

8.5 MONITOR / MWAIT

The hypervisor does not support the use of the MONITOR instruction but does have limited support for MWAIT. Partitions possessing the *CpuManagement* privilege (i.e. the root partition) may use MWAIT to set the logical processor's C-state if support for the instruction is present in hardware. Availability is indicated by the presence of a flag returned by the CPUID instruction for a hypervisor leaf (see section 2.4). Any attempt to use these instructions when the hypervisor does not indicate availability will result in a #UD fault.

8.6 System Management Mode

The hypervisor does not support or participate in the virtualization of system management mode within guest partitions. Physical system management interrupts are still handled normally by the system's hardware and firmware and is opaque to the hypervisor.

8.7 Time Stamp Counter

The time stamp counter (TSC) is virtualized for each virtual processor. Generally, the TSC value continues to run while a virtual processor is suspended.

Seamless TSC virtualization is not feasible on the x64 architecture. TSC virtualization is typically implemented through a simple TSC bias (an offset added to the logical processor's TSC). Attempts will be made by the hypervisor to prevent the TSC from jumping forward or backward as a virtual processor is scheduled on different logical processors. However, it cannot compensate for the situation where the TSC for a logical processor is set to zero by an SMI handler. Furthermore, the TSC increment rate may slow down or speed up depending on thermal or performance throttling, over which the hypervisor has no control.

Guest software should only use the TSC for measuring short durations. Even when using the TSC in this simple way, algorithms should be resilient to sudden jumps forward or backward in the TSC value.

8.8 Memory Accesses

The behavior of instructions that access memory may differ from the behavior of the same instruction on a logical processor. This is the result of the hypervisor's physical memory virtualization mechanisms and of the existence of address ranges with special semantics (hypervisor call page or SynIC area). In a broad sense, this applies to all instructions because the processor fetches instructions from memory. However, it applies in particular to instructions with memory operands.

The following pseudo-code defines the different behaviors that can result from an access by a virtual processor to its partition's GPA space. The pseudo-code assumes that a GPA memory access has been performed directly (that is, an explicit memory operand) or indirectly (an implicit access) by a virtual processor. The access is one of three types: Read, Write, and Execute (Instruction Fetch).

```
if the address is within an overlay page
{
   if the access type is not allowed for the page
   {
      Generate #MC fault within guest
   }
   else
   {
      Perform access
   }
}
else if the address is within an unmapped GPA page
{
   if the partition is the root partition
   {
      Allow the access to proceed to identity-mapped SPA
   }
   else
   {
      Suspend VP and send message to parent (umapped GPA)
   }
}
else if the address is within a mapped GPA and
   the access type violates the mapping’s access rights
{
   if the partition is the root partition
   {
      Generate #MC fault in root
   }
   else
   {
      Suspend VP and send message to parent (GPA access right)
   }
}
else
{
   Memory access proceeds normally
}
```

8.9 I/O Port Accesses

The behavior of instructions that access I/O ports may differ from the behavior of the same instruction on a logical processor. This is the result of the hypervisor’s processor intercept mechanism.

The following pseudo-code defines the different behaviors that can result from an access by a virtual processor to I/O ports (through the instructions IN, OUT, INS, or OUTS). Note that each of these instructions has an operand size of 1, 2, or 4 bytes. As such, one or more I/O ports are effectively accessed.

```
if any of the accessed ports is virtualized
   by the hypervisor for this access type
{
      Access is emulated
}
else if the I/O Port intercept is installed
{
      Suspend VP and send message to parent (I/O Port Intercept)
}
else if the partition is a non-root partition
{
      Discard writes; return all bits set for reads
}
else
{
      I/O port access proceeds normally
}
```

8.10 MSR Accesses

The behavior of instructions that access MSRs may differ from the behavior of the same instruction on a logical processor. This is the result of the hypervisor's processor intercept mechanism.

```
The following pseudo-code defines the different behaviors that can result
from an access by a virtual processor to MSRs.
if the MSR is virtualized by the hypervisor AND
   the partition possesses the privilege required by the MSR
{
      Access is emulated
}
else if the MSR intercept is installed
{
      Suspend VP and send message to parent (MSR intercept)
}
else
{
      Generate #GP fault within the guest
}
```

The hypervisor may virtualize MSRs as part of its interface with the guest. A summary of these can be found in section 19.

For those MSRs that are not virtualized by the hypervisor, internal security policy may require that certain fields within certain MSRs remain unmodified, are explicitly set for or hidden from the guest. In these cases, the access will appear to succeed from the guest's perspective, but the value actually written or read may not match the underlying physical MSR value.

8.11 CPUID Execution

The behavior of the CPUID instruction may differ from the behavior of the same instruction on a logical processor.

The following pseudo-code defines the different behaviors that can result from the execution of a CPUID instruction by a virtual processor.

```
if an intercept has been set for the CPUID instruction for
      the index specified in EAX
{
      Suspend VP and send message to parent (CPUID Intercept)
}
else
{
      CPUID instruction returns information as dictated by the
            logical processor and the hypervisor
}
```

The hypervisor may override the standard CPUID information returned by the logical processor.

NOTE

The hypervisor does not attempt to dictate a processor selection or to standardize on a particular processor model. The manipulation of various CPUID output is used to accommodate processor specifics or to reflect limitations on the partition's accessibility or privilege to use certain processor features.

8.12 Non-Privileged Instruction Execution Prevention (NPIEP)

Non-Privileged Instruction Execution (NPIEP) is a feature that limits the use of certain instructions by user-mode code. Specifically, when enabled, this feature can block the execution of the SIDT, SGDT, SLDT, and STR instructions. Execution of these instructions results in a #GP fault.

This feature must be configured on a per-VP basis using HV_X64_MSR_NPIEP_CONFIG_CONTENTS:

```
union
{
    UINT64 AsUINT64;
    struct
    {
        // These bits enable instruction execution prevention for specific
        // instructions.

        UINT64 PreventSgdt:1;
        UINT64 PreventSidt:1;
        UINT64 PreventSldt:1;
        UINT64 PreventStr:1;

        UINT64 Reserved:60;
    };
} HV_X64_MSR_NPIEP_CONFIG_CONTENTS;
```

8.13 Exceptions

The hypervisor’s intercept redirection mechanism allows a parent partition to intercept processor-generated exceptions in the virtual processors of a child partition. When the intercept message is delivered, the virtual processor will be in a restartable state (that is, the instruction pointer will point to the instruction that generated the exception).

Exception intercepts are checked before multiple exceptions are combined into a double fault or a triple fault. For example, if an exception intercept is installed on the #NP exception and a #NP exception occurs during the delivery of a #GP exception, the #NP exception intercept is triggered. Conversely, if no intercept was installed on the #NP exception, the nested #NP exception is converted into a double fault, which will trigger an intercept on the #DF if such an intercept was installed.

Note that exception intercepts do not occur for software-generated interrupts (that is, through the instructions INT, INTO, INT 3, and ICEBKPT).

The order in which exceptions are detected and reported by the processor depends on the instruction. For example, many instructions can generate multiple exceptions and the order in which these exceptions are detected is well defined.

The way in which exception intercepts interact with other intercept types also depends on the instruction. For example, an IN instruction may generate a #GP exception intercept before an I/O port intercept, and a RDMSR instruction may generate an MSR intercept before a #GP exception intercept. For details on the order of intercept delivery, consult the documentation for Intel’s and AMD’s processor virtualization extensions.

9 Virtual MMU and Caching

The virtual machine interface exposed by each partition includes a memory management unit (MMU). The virtual MMU exposed by hypervisor partitions is generally compatible with existing MMUs.

The hypervisor also supports guest-defined memory cacheability attributes for pages mapped into a partition's GVA space.

9.1 Virtual MMU Overview

Virtual processors expose virtual memory and a virtual TLB (translation look-aside buffer), which caches translations from virtual addresses to (guest) physical addresses. As with the TLB on a logical processor, the virtual TLB is a non-coherent cache, and this non-coherence is visible to guests. The hypervisor exposes operations to flush the TLB. Guests can use these operations to remove potentially inconsistent entries and make virtual address translations predictable.

9.1.1 **Compatibility**

The virtual MMU exposed by the hypervisor is generally compatible with the physical MMU found within an x64 processor. The following guest-observable differences exist:

- The CR3.PWT and CR3.PCD bits may not be supported in some hypervisor implementations. On such implementations, any attempt by the guest to set these flags through a MOV to CR3 instruction or a task gate switch will be ignored. Attempts to set these bits programmatically through HvSetVpRegisters or HvSwitchVirtualAddressSpace may result in an error.
- The PWT and PCD bits within a leaf page table entry (for example, a PTE for 4-K pages and a PDE for large pages) specify the cacheability of the page being mapped. The PAT, PWT, and PCD bits within non-leaf page table entries indicate the cacheability of the next page table in the hierarchy. Some hypervisor implementations may not support these flags. On such implementations, all page table accesses performed by the hypervisor are done by using write-back cache attributes. This affects, in particular, *accessed* and *dirty* bits written to the page table entries. If the guest sets the PAT, PWT, or PCD bits within non-leaf page table entries, an "unsupported feature" message may be generated when a virtual processor accesses a page that is mapped by that page table.
- The CR0.CD (cache disable) bit may not be supported in some hypervisor implementations. On such implementations, the CR0.CD bit must be set to 0. Any attempt by the guest to set this flag through a MOV to CR0 instruction will be ignored. Attempts to set this bit programmatically through HvSetVpRegisters will result in an error.
- The PAT (page address type) MSR is a per-VP register. However, when all the virtual processors in a partition set the PAT MSR to the same value, the new effect becomes a partition-wide effect.
- For reasons of security and isolation, the INVD instruction will be virtualized to act like a WBINVD instruction, with some differences. For security purposes, CLFLUSH should be used instead.

9.1.2 **Legacy TLB Management Operations**

The x64 architecture provides several ways to manage the processor's TLBs. The following mechanisms are virtualized by the hypervisor:

- The INVLPG instruction invalidates the translation for a single page from the processor's TLB. If the specified virtual address was originally mapped as a 4-K page, the translation for this page is removed from the TLB. If the specified virtual address was originally mapped as a "large page" (either 2 MB or 4 MB, depending on the MMU mode), the translation for the entire large page is removed from the TLB. The INVLPG instruction flushes both global and non-global translations. Global translations are defined as those which have the "global" bit set within the page table entry.
- The MOV to CR3 instruction and task switches that modify CR3 invalidate translations for all non-global pages within the processor's TLB.
- A MOV to CR4 instruction that modifies the CR4.PGE (global page enable) bit, the CR4.PSE (page size extensions) bit, or CR4.PAE (page address extensions) bit invalidates all translations (global and non-global) within the processor's TLB.

Note that all of these invalidation operations affect only one processor. To invalidate translations on other processors, software must use a software-based "TLB shoot-down" mechanism (typically implemented by using inter-process interrupts).

9.1.3 **Virtual TLB Enhancements**

In addition to supporting the legacy TLB management mechanisms described earlier, the hypervisor also supports a set of enhancements that enable a guest to manage the virtual TLB more efficiently.

These enhanced operations can be used interchangeably with legacy TLB management operations. On some systems (those with sufficient virtualization support in hardware), the legacy TLB management instructions may be faster for local or remote (cross-processor) TLB invalidation. Guests who are interested in optimal performance should use the CPUID leaf 0x40000004 to determine which behaviors to implement using hypercalls:

- *UseHypercallForAddressSpaceSwitch*: If this flag is set, the caller should assume that it's faster to use HvSwitchAddressSpace to switch between address spaces. If this flag is clear, a MOV to CR3 instruction is recommended.
- *UseHypercallForLocalFlush*: If this flag is set, the caller should assume that it's faster to use hypercalls (as opposed to INVLPG or MOV to CR3) to flush one or more pages from the virtual TLB.
- *UseHypercallForRemoteFlushAndLocalFlushEntire*: If this flag is set, the caller should assume that it's faster to use hypercalls (as opposed to using guest-generated inter-processor interrupts) to flush one or more pages from the virtual TLB.

9.1.4 **Restrictions on TLB Flushes**

When a virtual processor generates an intercept—especially those associated with memory accesses, software running within the parent or higher VTL may want to complete the intercepted instruction in software. This instruction completion logic will need to emulate the address translation normally performed by the processor's MMU. If a TLB flush request is executed on another virtual processor during instruction completion, incorrect behavior can result. For example, the second virtual processor could clear the dirty bit within the guest's page table and then request a TLB flush. If the instruction completion software modifies the contents of this page after the TLB flush request has been completed, the operating system running within the partition will not be notified of the page modification, and data corruption can occur.

To prevent this situation, the hypervisor provides a way to inhibit TLB flush hypercalls until intercept processing is complete. When a memory intercept message is generated by the hypervisor, the "TLB Flush Inhibit" bit (TlbFlushInhibit) will consequently be set. Any attempt to flush the TLB with a hypercall will place the caller's virtual processor in a suspended state. The instruction pointer will not be incremented past the instruction that invoked the hypercall. After the memory intercept routine performs instruction completion, it should clear the TlbFlushInhibit bit of the HvRegisterInterceptSuspend register. This resumes virtual processors that were suspended when they attempted to flush the TLB while the bit was set. Since the instruction pointer has not been incremented, the flush hypercall will automatically be re-executed. If the TlbFlushInhibit bit is clear, the hypercall will complete the flush normally.

9.2 Memory Cache Control Overview

9.2.1 Cacheability Settings

The hypervisor supports guest-defined cacheability settings for pages mapped within the guest's GVA space. For a detailed description of available cacheability settings and their meanings, refer to the Intel or AMD documentation.

When a virtual processor accesses a page through its GVA space, the hypervisor honors the cache attribute bits (PAT, PWT, and PCD) within the guest page table entry used to map the page. These three bits are used as an index into the partition's PAT (page address type) register to look up the final cacheability setting for the page.

Pages accessed directly through the GPA space (for example, when paging is disabled because CR0.PG is cleared) use a cacheability defined by the MTRRs. If the hypervisor implementation doesn't support virtual MTRRs, WB cacheability is assumed.

9.2.2 Mixing Cache Types between a Partition and the Hypervisor

Guests should be aware that some pages within its GPA space may be accessed by the hypervisor. The following list, while not exhaustive, provides several examples:

- A page that contains input or output parameters for a hypercall
- All overlay pages including the hypercall page, SynIC SIEF and SIM pages, and stats pages

The hypervisor always performs accesses to hypercall parameters and overlay pages by using the WB cacheability setting.

9.3 Virtual MMU Data Types

9.3.1 Virtual Address Spaces

The hypervisor introduces the concept of a virtual address space. The guest uses virtual address spaces to define the mapping between guest virtual addresses (GVAs) and guest physical addresses (GPAs). The guest OS can decide how and where to use virtual address spaces. In most OSs (including Microsoft Windows®), a different virtual address space is used for each process.

Virtual address spaces are identified by a caller-defined 64-bit ID value. On x64 implementations of the hypervisor, this value is the same as the value within the virtual processor's CR3 register, which points to the guest's page table structures.

```
typedef UINT64 HV_ADDRESS_SPACE_ID;
```

9.3.2 **Virtual Address Flush Flags**

The hypervisor provides hypercalls that allow the guest to flush (that is, invalidate) entire virtual address spaces or portions of these address spaces. Behavior of the flush operation can be modified by using a set of flags, defined as follows:

```
typedef UINT32 HV_FLUSH_FLAGS;

#define HV_FLUSH_ALL_PROCESSORS                  0x00000001
#define HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES      0x00000002
#define HV_FLUSH_NON_GLOBAL_MAPPINGS_ONLY        0x00000004
#define HV_FLUSH_USE_EXTENDED_RANGE_FORMAT       0x00000008
```

9.3.3 **Cache Types**

Several structures include cache type fields. The following encodings are defined:

```
typedef enum
{
   HvCacheTypeX64Uncached     = 0,
   HvCacheTypeX64WriteCombining     = 1,
   HvCacheTypeX64WriteThrough = 4,
   HvCacheTypeX64WriteProtected     = 5,
   HvCacheTypeX64WriteBack    = 6
} HV_CACHE_TYPE;
```

9.3.4 **Virtual Address Translation Types**

```
The call HvTranslateVirtualAddress takes a collection of input control flags
and returns a result code and a collection of output flags. The input control
flags are defined as follows:
typedef UINT64 HV_TRANSLATE_GVA_CONTROL_FLAGS;

#define HV_TRANSLATE_GVA_VALIDATE_READ    0x0001
#define HV_TRANSLATE_GVA_VALIDATE_WRITE   0x0002
#define HV_TRANSLATE_GVA_VALIDATE_EXECUTE 0x0004
#define HV_TRANSLATE_GVA_PRIVILEGE_EXEMPT 0x0008
#define HV_TRANSLATE_GVA_SET_PAGE_TABLE_BITS    0x0010
#define HV_TRANSLATE_GVA_TLB_FLUSH_INHIBIT      0x0020
#define HV_TRANSLATE_GVA_CONTROL_MASK        (0x003F)
#define HV_TRANSLATE_GVA_INPUT_VTL_MASK      (0xFF00000000000000)
```

The returned result code is defined as follows:

```
typedef enum
{
   HvTranslateGvaSuccess                  = 0,

   // Translation failures
   HvTranslateGvaPageNotPresent           = 1,
   HvTranslateGvaPrivilegeViolation       = 2,
   HvTranslateGvaInvalidPageTableFlags    = 3,

   // GPA access failures
   HvTranslateGvaGpaUnmapped              = 4,
   HvTranslateGvaGpaNoReadAccess          = 5,
   HvTranslateGvaGpaNoWriteAccess         = 6,
   HvTranslateGvaGpaIllegalOverlayAccess  = 7,

   //
   // Intercept of the memory access by either
   // - a higher VTL
   // - a nested hypervisor (due to a violation of the nested page table)
   //
   HvTranslateGvaIntercept                = 8,} HV_TRANSLATE_GVA_RESULT_CODE;

typedef enum HV_TRANSLATE_GVA_RESULT_CODE
   *PHV_TRANSLATE_GVA_RESULT_CODE;

typedef struct
{
   HV_TRANSLATE_GVA_RESULT_CODE     ResultCode;
   UINT32   CacheType:8;
   UINT32   OverlayPage:1;
   UINT32   Reserved3:23;
} HV_TRANSLATE_GVA_RESULT;

typedef struct
{
    HV_TRANSLATE_GVA_RESULT_CODE ResultCode;
    UINT32 CacheType : 8;
    UINT32 OverlayPage : 1;
    UINT32 Reserved : 23;
    HV_X64_PENDING_EVENT EventInfo;

} HV_TRANSLATE_GVA_RESULT_EX;
```

9.4 Virtual MMU Interfaces

9.4.1 HvSwitchVirtualAddressSpace

The HvSwitchVirtualAddressSpace hypercall switches the calling virtual processor's virtual address space.

Wrapper Interface

```
HV_STATUS
HvSwitchVirtualAddressSpace(
      __in HV_ADDRESS_SPACE_ID      AddressSpace
      );
```

Native Interface

HvSwitchVirtualAddressSpace [fast]	
	Call Code = 0x0001
➡ Input Parameters	
0	AddressSpace (8 bytes)

Description

For x64 implementations of the hypervisor, this call also updates the CR3 register. However, unlike a MOV to CR3 instruction, this hypercall does not have the side-effect of flushing the virtual processor's TLB.

This hypercall, unlike most, operates implicitly in the context of the calling partition and virtual processor.

Input Parameters

AddressSpace specifies a new address space ID (a new CR3 value).

Output Parameters

None.

Restrictions

None.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	The specified address space ID is not a valid CR3 value.
	One or more reserved bits in the specified address space ID (as defined by the x64 architecture) were set.

9.4.2 HvFlushVirtualAddressSpace

The HvFlushVirtualAddressSpace hypercall invalidates all virtual TLB entries that belong to a specified address space.

Wrapper Interface

```
HV_STATUS
HvFlushVirtualAddressSpace(
      __in HV_ADDRESS_SPACE_ID      AddressSpace,
      __in HV_FLUSH_FLAGS     Flags,
      __in UINT64 ProcessorMask
      );
```

Native Interface

HvFlushVirtualAddressSpace	
	Call Code = 0x0002
➡ Input Parameters	
0	AddressSpace (8 bytes)
8	Flags (8 bytes)
16	ProcessorMask (8 bytes)

Description

The virtual TLB invalidation operation acts on one or more processors.

If the guest has knowledge about which processors may need to be flushed, it can specify a processor mask. Each bit in the mask corresponds to a virtual processor index. For example, a mask of 0x0000000000000051 indicates that the hypervisor should flush only the TLB of virtual processors 0, 4, and 6. A virtual processor can determine its index by reading from MSR HV_X64_MSR_VP_INDEX.

The following flags can be used to modify the behavior of the flush:

- *HV_FLUSH_ALL_PROCESSORS* indicates that the operation should apply to all virtual processors within the partition. If this flag is set, the *ProcessorMask* parameter is ignored.
- *HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES* indicates that the operation should apply to all virtual address spaces. If this flag is set, the *AddressSpace* parameter is ignored.
- *HV_FLUSH_NON_GLOBAL_MAPPINGS_ONLY* indicates that the hypervisor is required only to flush page mappings that were not mapped as “global” (that is, the x64 “G” bit was set in the page table entry). Global entries may be (but are not required to be) left unflushed by the hypervisor.

All other flags are reserved and must be set to zero.

This call guarantees that by the time control returns back to the caller, the observable effects of all flushes on the specified virtual processors have occurred.

If a target virtual processor’s TLB requires flushing and that virtual processor’s TLB is currently “locked”, the caller’s virtual processor is suspended. When the caller’s virtual processor is “unsuspended”, the hypercall will be reissued.

Input Parameters

AddressSpace specifies an address space ID (a CR3 value).

Flags specifies a set of flag bits that modify the operation of the flush.

ProcessorMask specifies a processor mask indicating which processors should be affected by the flush operation.

Output Parameters

None.

Restrictions

None.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	The specified address space ID is not a valid CR3 value and the ”flush all virtual address spaces” flag was not specified.
	One or more reserved bits in the specified address space ID (as defined by the x64 architecture) were set.
	One or more reserved bits within the flags register are set.
	All of the bits in the processor bit mask are set to zero, and the "flush all processors" flag was not specified.

9.4.3 **HvFlushVirtualAddressSpaceEx**

The HvFlushVirtualAddressSpaceEx hypercall is similar to HvFlushVirtualAddressSpace, but can take a variably-sized sparse VP set as an input.

The following checks should be used to infer the availability of this hypercall:

1. ExProcessorMasks must be indicated via CPUID leaf 0x40000004 (see 2.4.5)

Wrapper interface

```
HV_STATUS
HvFlushVirtualAddressSpaceEx(
   __in HV_ADDRESS_SPACE_ID   AddressSpace,
   __in HV_FLUSH_FLAGS  Flags,
   __in HV_VP_SET ProcessorSet
   );
```

Native Interface

HvFlushVirtualAddressSpaceEx	
	Call Code = 0x0013
➡ Input Parameters	
0	AddressSpace (8 bytes)
8	Flags (8 bytes)
16	ProcessorSet (Variably sized)

Description

See HvFlushVirtualAddressSpace.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	A parameter is invalid.

9.4.4 **HvFlushVirtualAddressList**

The HvFlushVirtualAddressList hypercall invalidates portions of the virtual TLB that belong to a specified address space.

Wrapper Interface

```
HV_STATUS
HvFlushVirtualAddressList(
      __in    HV_ADDRESS_SPACE_ID   AddressSpace,
      __in    HV_FLUSH_FLAGS  Flags,
      __in    UINT64    ProcessorMask,
      __inout PUINT32   GvaCount,
      __in_ecount(GvaCount)
              PCHV_GVA  GvaRangeList
      );
```

Native Interface

HvFlushVirtualAddressList [rep]	
	Call Code = 0x0003
➡ Input Parameter Header	
0	AddressSpace (8 bytes)
8	Flags (8 bytes)
16	ProcessorMask (8 bytes)
➡ Input List Element	
0	GvaRange (8 bytes)

Description

The virtual TLB invalidation operation acts on one or more processors.

If the guest has knowledge about which processors may need to be flushed, it can specify a processor mask. Each bit in the mask corresponds to a virtual processor index. For example, a mask of 0x0000000000000051 indicates that the hypervisor should flush only the TLB of virtual processors 0, 4 and 6.

The following flags can be used to modify the behavior of the flush:

- *HV_FLUSH_ALL_PROCESSORS* indicates that the operation should apply to all virtual processors within the partition. If this flag is set, the *ProcessorMask* parameter is ignored.
- *HV_FLUSH_ALL_VIRTUAL_ADDRESS_SPACES* indicates that the operation should apply to all virtual address spaces. If this flag is set, the *AddressSpace* parameter is ignored.
- *HV_FLUSH_NON_GLOBAL_MAPPINGS_ONLY* does not make sense for this call and is treated as an invalid option.

All other flags are reserved and must be set to zero.

This call takes a list of GVA ranges. Each range has a base GVA. Because flushes are performed with page granularity, the bottom 12 bits of the GVA can be used to define a range length. These bits encode the number of additional pages (beyond the initial page) within the range. This allows each entry to encode a range of 1 to 4096 pages.

A GVA that falls within a “large page” mapping (2MB or 4MB) will cause the entire large page to be flushed from the virtual TLB.

This call guarantees that by the time control returns back to the caller, the observable effects of all flushes on the specified virtual processors have occurred.

Invalid GVAs (those that specify addresses beyond the end of the partition’s GVA space) are ignored.

If a target virtual processor’s TLB requires flushing and that virtual processor is inhibiting TLB flushes, the caller’s virtual processor is suspended. When TLB flushes are no longer inhibited, the virtual processor is “unsuspended” and the hypercall will be reissued.

Input Parameters

AddressSpace specifies an address space ID (a CR3 value).

Flags specifies a set of flag bits that modify the operation of the flush.

ProcessorMask specifies a processor mask indicating which processors should be affected by the flush operation.

GvaRange specifies a guest virtual address range.

Output Parameters

None.

Restrictions

None.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	The specified address space ID is not a valid CR3 value and the "flush all virtual address spaces” flag was not specified.
	One or more reserved bits in the specified address space ID (as defined by the x64 architecture) were set.
	One or more reserved bits within the flags register are set.
	All of the bits in the processor bit mask are set to zero, and the "flush all processors" flag was not specified.

9.4.5 **HvFlushVirtualAddressListEx**

The HvFlushVirtualAddressListEx hypercall is similar to HvFlushVirtualAddressList, but can take a variably-sized sparse VP set as an input.

The following checks should be used to infer the availability of this hypercall:

1. ExProcessorMasks must be indicated via CPUID leaf 0x40000004 (see 2.4.5)

Wrapper interface

```
HV_STATUS
HvFlushVirtualAddressListEx(
   __in    HV_ADDRESS_SPACE_ID      AddressSpace,
   __in    HV_FLUSH_FLAGS           Flags,
   __in    HV_VP_SET                ProcessorSet,
   __inout PUINT32                  GvaCount,
   __in_ecount(GvaCount) PCHV_GVA   GvaRangeList
   );
```

Native Interface

HvFlushVirtualAddressList [rep]	
	Call Code = 0x0014
➡ Input Parameter Header	
0	AddressSpace (8 bytes)
8	Flags (8 bytes)
	ProcessorSet (Variably sized)
➡ Input List Element	
0	GvaRange (8 bytes)

Description

See HvFlishVirtualAddressList.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	A parameter is invalid.

9.4.6 **HvTranslateVirtualAddress**

The HvTranslateVirtualAddress hypercall attempts to translate a specified GVA page number into a GPA page number.

Wrapper Interface

```
HV_STATUS
HvTranslateVirtualAddress(
      __in  HV_PARTITION_ID   PartitionId,
      __in  HV_VP_INDEX VpIndex,
      __in  HV_TRANSLATE_GVA_CONTROL_FLAGS      ControlFlags,
      __in  HV_GVA_PAGE_NUMBER      GvaPage,
      __out PHV_TRANSLATE_GVA_RESULT      TranslationResult,
      __out PHV_GPA_PAGE_NUMBER     GpaPage
      );
```

Native Interface

HvTranslateVirtualAddress		
	Call Code = 0x0052	
🡆 Input Parameters		
0	PartitionId (8 bytes)	
8	VpIndex (4 bytes)	Padding (4 bytes)
16	ControlFlags (8 bytes)	
24	GvaPage (8 bytes)	
🡄 Output Parameters		
0	TranslationResult (8 bytes)	
8	GpaPage (8 bytes)	

Description

The translation considers the current modes and state of the specified virtual processor as well as the guest page tables.

The caller must specify whether the intended access is to read, write or execute by setting the appropriate control flags. Combinations of these access types are possible. Several other translation options are also available.

- *HV_TRANSLATE_GVA_PRIVILEGE_EXEMPT*: Indicates that the access should be performed as though the processor was running at a privilege level zero rather than the current privilege level.
- *HV_TRANSLATE_GVA_SET_PAGE_TABLE_BITS*: Indicates that the routine should set the dirty and accessed bits within the guest's page tables if appropriate for the access type. The dirty bit will only be set if *HV_TRANSLATE_GVA_VALIDATE_WRITE* is also specified. If the caller has

requested that accessed and dirty bits be set as part of the table walk, these bits are set as the walk occurs. If a walk is aborted, the accessed and dirty bits that were already set are not restored to their previous values.

- *HV_TRANSLATE_GVA_TLB_FLUSH_INHIBIT*: Indicates that the *TlbFlushInhibit* flag in the virtual processor's HvRegisterInterceptSuspend register should be set as a consequence of a successful return. This prevents other virtual processors associated with the target partition from flushing the stage 1 TLB of the specified virtual processor until after the *TlbFlushInhibit* flag is cleared (see 9.1.4).

If paging is disabled in the virtual processor (that is, CR0.PG is clear), then no page tables are consulted, and translation success is guaranteed.

If paging is enabled in the virtual processor (that is, CR0.PG is set), then a page table walk is performed. The call uses the current state of the virtual processor to determine whether to perform a two-level, three-level, or four-level page table walk. The caller may not assume that the walk is coherent with the hardware TLB state.

During the page table walk, a number of conditions can arise that cause the walk to be terminated:

- A table entry is marked "not present" or the GVA is beyond the range permitted for the paging mode. In this case, *HvTranslateGvaPageNotPresent* is returned.
- A privilege violation is detected based on the access type (read, write, execute) or on the current privilege level. In this case, *HvTranslateGvaPrivilegeViolation* is returned.
- A reserved bit is set within a table entry. In this case, *HvTranslateGvaInvalidPageTableFlags* is returned.
- A page table walk can also be terminated if one of the guest's page table pages cannot be accessed. This can occur in one of the following situations: The GPA is unmapped. In this case, *HvTranslateGvaGpaUnmapped* is returned.
- The GPA mapping's access rights indicate that the page is not readable. In this case, *HvTranslateGvaGpaNoReadAccess* or *HvTranslateGvaGpaNoWriteAccess* is returned.
- The access targets an overlay page that doesn't allow reads or writes. In this case, *HvTranslateGvaGpaIllegalOverlayAccess* is returned.
- If any of these GPA access failures are reported, the *GpaPage* output parameter is used to indicate which GPA page could not be accessed.

If no translation error occurs, *HvTranslateGvaSuccess* is returned. In this case, the *GpaPage* output parameter is used to report the resulting translation, and the associated *CacheType* and *OverlayPage* fields are set appropriately. The *CacheType* field indicates the effective cache type used by the virtual processor to access the translated virtual address. The *OverlayPage* field indicates whether the translated GPA accesses an overlay page owned by the hypervisor. Callers can use this information to determine whether memory accesses performed by the virtual processor would have accessed a mapped GPA page or an overlay page.

If the caller has requested that accessed and dirty bits be set as part of the table walk, then these bits are set as the walk occurs. If a walk is aborted, then the accessed and dirty bits that were already set are not restored to their previous values.

The reported cache type considers all of the state of the virtual processor including the current virtual PAT register settings and (if supported by the hypervisor implementation) the value of the MTRR MSRs and CR0.CD.

If the call returns HV_STATUS_SUCCESS, the output parameter *TranslationResult* is valid. The caller must consult the result code and results flags to determine whether the *GpaPage* parameter is valid.

Input Parameters

PartitionId specifies a partition.

VpIndex specifies a virtual processor index.

ControlFlags specifies a set of flag bits that modify the behavior of the translation.

GvaPage specifies a guest virtual address page number.

Output Parameters

TranslationResult specifies information about the translation including the result code and flags.

GpaPage specifies the translated GPA (if the result code is HvTranslateGvaSuccess) or the address of a GPA access failure (if the result code is HvTranslateGvaGpaUnmapped, HvTranslateGvaGpaNoReadAccess, HvTranslateGvaGpaNoWriteAccess, or HvTranslateGvaGpaIllegalOverlayAccess). For other result codes, this return parameter is invalid.

Restrictions

- The partition specified by *PartitionId* must be in the “active” state.
- The caller must be the parent of the partition specified by *PartitionId*.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_VP_INDEX	The specified VP index does not reference a virtual processor within the specified partition.
HV_STATUS_INVALID_PARAMETER	All three of the control flags HV_TRANSLATE_GVA_VALIDATE_READ, HV_TRANSLATE_GVA_VALIDATE_WRITE, and HV_TRANSLATE_GVA_VALIDATE_EXECUTE are cleared. At least one of these must be set.
	One or more reserved bits in the specified control flags are set.

Status code	Error condition
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.
HV_STATUS_INVALID_VP_STATE	A virtual processor is not in the correct state for the performance of the indicated operation.

9.4.7 HvExtCallGetBootZeroedMemory

Hyper-V allocates zero-filled pages to a VM at creation time. The HvExtCallGetBootZeroedMemory hypercall can be used to query which GPA pages were zeroed by Hyper-V during creation. This can prevent the guest memory manager from having to redundantly zero GPA pages, which can reduce utilization and increase performance.

This is an extended hypercall; its availability must be queried using HvExtCallQueryCapabilities.

Wrapper Interface

```
HV_STATUS
HvExtCallGetBootZeroedMemory(
      __out UINT64      StartGpa,
      __out UINT64      PageCount
      );
```

Native Interface

HvExtCallGetBootZeroedMemory	
	Call Code = 0x8002
🡄 Output Parameters	
0	StartGpa (8 bytes)
8	PageCount (8 bytes)

Input Parameters

None.

Output Parameters

StartGpa – the GPA address where the zeroed memory region begins.

PageCount – the number of pages included in the zeroed memory region.

Restrictions

- The availability of this hypercall must be queried using the HvExtCallQueryCapabilities.

10 Virtual Interrupt Control

10.1 Overview

The hypervisor virtualizes interrupt delivery to virtual processors. This is done through the use of a synthetic interrupt controller (SynIC) which is an extension of a virtualized local APIC; that is, each virtual processor has a local APIC instance with the SynIC extensions. These extensions provide a simple inter-partition communication mechanism which is described in the following chapter.

Interrupts delivered to a partition fall into two categories: external and internal. External interrupts originate from other partitions or devices, and internal interrupts originate from within the partition itself.

External interrupts are generated in the following situations:

- A physical hardware device generates a hardware interrupt.
- A parent partition asserts a virtual interrupt (typically in the process of emulating a hardware device).
- The hypervisor delivers a message (for example, due to an intercept) to a partition.
- Another partition posts a message.
- Another partition signals an event.

Internal interrupts are generated in the following situations:

- A virtual processor accesses the APIC interrupt command register (ICR).
- A synthetic timer expires.

10.2 Local APIC

The SynIC is a superset of a local APIC. The interface to this APIC is given by a set of 32-bit memory mapped registers. This local APIC (including the behavior of the memory mapped registers) is generally compatible with the local APIC on P4/Xeon systems as described in Intel’s documentation.

10.2.1 **Local APIC Virtualization**

The hypervisor’s local APIC virtualization may deviate from physical APIC operation in the following minor ways:

- On physical systems, the IA32_APIC_BASE MSR can be different for each processor in the system. The hypervisor may require that this MSR contains the same value for all virtual processors within a partition. As such, this MSR may be treated as a partition-wide value. If a virtual processor modifies this register, the value may effectively propagate to all virtual processors within the partition.
- The IA32_APIC_BASE MSR defines a “global enable” bit for enabling or disabling the APIC. The virtualized APIC may always be enabled. If so, this bit will always be set to 1.
- The hypervisor’s local APIC may not be able to generate virtual SMIs (system management interrupts).

- The hypervisor may allow accesses only to the APIC's memory-mapped registers to be performed by one of the instructions in section 10.2.2. Furthermore, it may allow only accesses that are four bytes in size and aligned to four-byte boundaries. In such cases, if an unsupported access is attempted, the virtual processor will be suspended, and an unsupported feature error message will be delivered to the partition's parent.
- If multiple virtual processors within a partition are assigned identical APIC IDs, behavior of targeted interrupt delivery is boundedly undefined. That is, the hypervisor is free to deliver the interrupt to just one virtual processor, all virtual processors with the specified APIC ID, or no virtual processors. This situation is considered a guest programming error.
- Some of the memory mapped APIC registers may be accessed by way of virtual MSRs.
- The hypervisor may not allow a guest to modify its APIC IDs.

The remaining parts of this section describe only those aspects of SynIC functionality that are extensions of the local APIC.

10.2.2 **Local APIC Memory-mapped Accesses**

The hypervisor emulates accesses to memory-mapped registers within the virtualized local APIC. However, only certain instruction forms are supported, and use of other forms will result in #GP. Compatible guests should access only the local APIC registers by using the following instruction forms:

Opcode	Instruction	Notes
89 /r	MOV m32,r32	m32 must be 4-byte aligned.
8B /r	MOV r32,m32	m32 must be 4-byte aligned.
A1	MOV EAX,moffs32	moffs32 must be 4-byte aligned.
A3	MOV moffs32,EAX	moffs32 must be 4-byte aligned.
C7 /0	MOV m32,imm32	m32 must be 4-byte aligned.
FF /6	PUSH m32	m32 must be 4-byte aligned.

10.2.3 **Local APIC MSR Accesses**

The hypervisor provides accelerated MSR access to high usage memory mapped APIC registers. These are the TPR, EOI, and the ICR registers. The ICR low and ICR high registers are combined into one MSR.

MSR Address	Register Name	Function
0x40000070	HV_X64_MSR_EOI	Accesses the APIC EOI
0x40000071	HV_X64_MSR_ICR	Accesses the APIC ICR-high and ICR-low
0x40000072	HV_X64_MSR_TPR	Access the APIC TPR

For performance reasons, the guest operating system should follow the hypervisor recommendation for the usage of the APIC MSRs (see 2.4)

10.2.3.1 **EOI Register**

63:32	31:0
Ignored	EOI value

Bits	Description	Attributes
63:32	RsvdZ (reserved, should be zero)	Write
31:0	EOI value	Write

This is a write-only register, and it sets a value into the APIC EOI register. Attempts to read from this register will result in a #GP fault.

10.2.3.2 **ICR Register**

63:32	31:0
ICR high	ICR low

Bits	Description	Attributes
63:32	ICR high value	Read/write
31:0	ICR low value	Read/write

The values of ICR high and ICR low are read from or written into the corresponding APIC ICR high and low registers.

10.2.3.3 **TPR Register**

63:8	7:0
RsvdZ	TPR value

Bits	Description	Attributes
63:8	RsvdZ (reserved, should be zero)	Read/write
7:0	TPR value	Read/write

The value of the APIC TPR register is read or written.

NOTE

This MSR is intended to accelerate access to the TPR in 32-bit mode guest partitions. 64-bit mode guest partitions should set the TPR by way of CR8.

10.3 Virtual Interrupts

10.3.1 **Virtual Interrupt Overview**

The hypervisor provides interfaces that allow a partition to send virtual interrupts to virtual processors. This is useful for emulating an IOAPIC or a legacy 8259 PIC (programmable interrupt controller).

10.3.2 **Virtual Interrupt Types**

To send a virtual interrupt, software must call HvAssertVirtualInterrupt and specify a virtual processor within the target partition or VTL. It must also specify the interrupt type that determines the behavior:

- *HvX64InterruptTypeNmi* generates a non-maskable interrupt on the specified processor.
- *HvX64InterruptTypeSmi* generates a system management interrupt on the specified processor.
- *HvX64InterruptTypeInit* generates an INIT interrupt on the specified processor.
- *HvX64InterruptTypeSipi* generates a start inter-processor interrupt. If the target processor is in wait-for-SIPI state, it causes the target processor to begin executing in real mode at an address determined by the SIPI vector as specified by the x64 architecture.
- *HvX64InterruptTypeFixed* generates a fixed interrupt latched into the local APIC's interrupt request register (IRR). A fixed interrupt can be edge-triggered or level-triggered. Withdrawing an edge-triggered interrupt does not clear the corresponding bit in the IRR. Withdrawing a level-triggered interrupt clears the corresponding bit in the IRR.
- *HvX64InterruptTypeLowestPriority* is like a fixed interrupt except that it is delivered only to the lowest-priority destination virtual processor.
- *HvX64InterruptTypeExtInt* generates a fixed level-triggered interrupt. The behavior is the same as with *HvX64InterruptTypeFixed,* with the following exceptions:
    - It is always directed at the boot processor, and
    - It can be used when the APIC is software disabled.
    - Regardless of whether the APIC is enabled or not, the PPR (process priority register) is not used in determining whether the interrupt will be serviced. It also requires the use of a separate hypercall, HvClearVirtualInterrupt, to clear an acknowledged interrupt before subsequent interrupts of this type can be asserted.

10.3.3 **Trigger Types**

Virtual interrupts are either edge-triggered or level-triggered. Edge-triggered interrupts are latched upon assertion and cannot be withdrawn. Level-triggered interrupts are not latched and can potentially be withdrawn by deasserting. The following table indicates, for each interrupt type, what the implicit interrupt trigger type is and whether a vector should be specified with the virtual interrupt.

Interrupt type	Vector applicable?	Trigger type
NMI	No	Edge
INIT	No	Edge
SIPI	Yes	Edge
Fixed	Yes	Edge or Level
Lowest Priority	Yes	Edge or Level
ExtINT	Yes	Level
SMI	Yes	Edge

Sometime after a virtual interrupt is asserted, it may be acknowledged by the virtual processor. Until then, level-triggered virtual interrupts can be deasserted by calling HvAssertVirtualInterrupt with vector HV_INTERRUPT_VECTOR_NONE or it can be re-asserted by calling HvAssertVirtualInterrupt. Deasserting an edge-triggered interrupt is unnecessary and has no effect.

10.3.4 **EOI Intercepts**

An intercept is defined for processor events (specifically, memory accesses) that indicate the EOI (end of interrupt) for a level-triggered fixed interrupt. An EOI intercept is the expected (eventual) response by the child to a parent asserting a level triggered interrupt using the HvAssertVirtualInterrupt hypercall. The intercept is delivered at the instruction boundary following the instruction that issued the EOI.

For performance reasons, it is desirable to reduce the number of EOI intercepts. Most EOI intercepts can be eliminated and done lazily if the guest OS leaves a marker when it performs an EOI. However, there are two cases for which EOI intercepts are strictly necessary.

- A level triggered interrupt is EOI'ed, since the hypervisor needs to either EOI the physical APIC (in case of the root partition) or send an EOI message (in case of a non-root partition) when the guest performs an EOI.
- A lower priority interrupt is pending, since the hypervisor needs to re-evaluate interrupts when the guest performs an EOI.

10.3.5 **EOI Assist**

One field in the virtual VP assist page (see 7.8.7) is the EOI Assist field. The EOI Assist field resides at offset 0 of the overlay page and is 32-bits in size. The format of the EOI assist field is as follows:

31:1	0
Reserved to Zero	No EOI Required

The OS performs an EOI by atomically writing zero to the EOI Assist field of the virtual VP assist page and checking whether the "No EOI required" field was previously zero. If it was, the OS must write to the HV_X64_APIC_EOI MSR thereby triggering an intercept into the hypervisor. The following code is recommended to perform an EOI:

```
lea   rcx, [VirtualApicAssistVa]
btr   [rcx], 0
jc    NoEoiRequired

mov   ecx, HV_X64_APIC_EOI
wrmsr

NoEoiRequired:
```

The hypervisor sets the “No EOI required” bit when it injects a virtual interrupt if the following conditions are satisfied:

- The virtual interrupt is edge-triggered, and
- There are no lower priority interrupts pending

If, at a later time, a lower priority interrupt is requested, the hypervisor clears the “No EOI required” such that a subsequent EOI causes an intercept.

In case of nested interrupts, the EOI intercept is avoided only for the highest priority interrupt. This is necessary since no count is maintained for the number of EOIs performed by the OS. Therefore only the first EOI can be avoided and since the first EOI clears the “No EOI Required” bit, the next EOI generates an intercept. However nested interrupts are rare, so this is not a problem in the common case.

Note that devices and/or the I/O APIC (physical or synthetic) need not be notified of an EOI for an edge-triggered interrupt – the hypervisor intercepts such EOIs only to update the virtual APIC state. In some cases, the virtual APIC state can be lazily updated – in such cases, the “NoEoiRequired” bit is set by the hypervisor indicating to the guest that an EOI intercept is not necessary. At a later instant, the hypervisor can derive the state of the local APIC depending on the current value of the “NoEoiRequired” bit.

Enabling and disabling this enlightenment can be done at any time independently of the interrupt activity and the APIC state at that moment. While the enlightenment is enabled, conventional EOIs can still be performed irrespective of the “No EOI required” value but they will not realize the performance benefit of the enlightenment.

10.4 Virtual Interrupt Data Types

10.4.1 Interrupt Types

Several virtual interrupt types are supported.

```
typedef enum
{
   HvX64InterruptTypeFixed                = 0x0000,
   HvX64InterruptTypeLowestPriority       = 0x0001,
   HvX64InterruptTypeNmi                  = 0x0004,
   HvX64InterruptTypeInit                 = 0x0005,
   HvX64InterruptTypeSipi                 = 0x0006,
   HvX64InterruptTypeExtInt               = 0x0007
} HV_INTERRUPT_TYPE;
```

10.4.2 Interrupt Control

The interrupt control specifies the type of the virtual interrupt, its destination mode and whether the virtual interrupt is edge or level triggered.

```
typedef struct
{
   HV_INTERRUPT_TYPE    InterruptType;
   UINT32   LevelTriggered:1;
   UINT32   LogicalDestinationMode:1;
   UINT32   Reserved:30;
} HV_INTERRUPT_CONTROL;
```

10.4.3 **Interrupt Vectors**

Interrupt vectors are represented by a 32-bit value. A special value is used to indicate “no interrupt vector” and is used by calls that indicate whether a previous interrupt was acknowledged.

```
typedef UINT32 HV_INTERRUPT_VECTOR;
typedef HV_INTERRUPT_VECTOR *PHV_INTERRUPT_VECTOR;


#define HV_INTERRUPT_VECTOR_NONE 0xFFFFFFFF
```

10.4.4 **MSI Entry**

```
typedef union
{

    UINT64 AsUINT64;

    struct
    {
        UINT32 Address;
        UINT32 Data;
    };

} HV_MSI_ENTRY;
```

Note: in the case of MSI multiple message enabled devices, each vector is retargeted independently. “Data” is the data value sent by the device when signaling the specific vector.

10.4.5 **Interrupt Source**

```
typedef enum
{
    HvInterruptSourceMsi = 1,

} HV_INTERRUPT_SOURCE;
```

10.4.6 **Interrupt Entry**

```
typedef struct
{
    HV_INTERRUPT_SOURCE InterruptSource;
    UINT32 Reserved;

    union
    {
        HV_MSI_ENTRY MsiEntry;
        UINT64 Data;
    };

} HV_INTERRUPT_ENTRY;
```

10.4.7 **Device Interrupt Target**

```
typedef struct
{
    HV_INTERRUPT_VECTOR Vector;
    UINT32 Flags;

    union
    {
        UINT64 ProcessorMask;
        UINT64 ProcessorSet[];
    };

} HV_DEVICE_INTERRUPT_TARGET;
```

"Flags" supplies optional flags for the interrupt target:

```
#define HV_DEVICE_INTERRUPT_TARGET_MULTICAST        1
#define HV_DEVICE_INTERRUPT_TARGET_PROCESSOR_SET    2
```

"Multicast" indicates that the interrupt is sent to all processors in the target set. By default, the interrupt is sent to an arbitrary single processor in the target set.

10.5 Virtual Interrupt Interfaces

10.5.1 **HvAssertVirtualInterrupt**

The HvAssertVirtualInterrupt hypercall requests a virtual interrupt to be presented to the specified virtual processor(s).

Wrapper Interface

```
HV_STATUS
HvAssertVirtualInterrupt(
   __in  HV_PARTITION_ID      DestinationPartition,
   __in  HV_INTERRUPT CONTROL InterruptControl,
   __in  UINT64               DestinationAddress,
   __in  HV_INTERRUPT_VECTOR  RequestedVector
   __in  HV_VTL               TargetVtl
   __in  UINT8                Reservedz0
   __in  UINT16               ReservedZ1
   );
```

Native Interface

HvAssertVirtualInterrupt			
	Call Code = 0x0094		
➡ Input Parameters			
0	DestinationPartition (8 bytes)		
8	InterruptControl (8 bytes)		
16	DestinationAddress (8 bytes)		
24	RequestedVector (4 bytes)	TargetVtl (1 byte)	Padding (3 bytes)

Description

For information on virtual interrupts, see 10.2.3.

If the call is made twice in a row with the same interrupt type specified in the *InterruptControl* parameter, the behavior depends upon whether or not the first interrupt was acknowledged by the virtual processor before the second call is made.

If the first interrupt has already been acknowledged, then the second call is treated as a new assertion.

If the first interrupt has not yet been acknowledged, then the second call supersedes the previous assertion with the new vector. If the second call specifies the vector HV_INTERRUPT_VECTOR_NONE, then the call acts as a deassertion.

The behavior of this call differs for interrupts of type *HvX64InterruptTypeExtInt* in the following ways:

This interrupt type is always targeted at the boot processor. The boot processor is identified by a virtual processor index of zero. The *DestinationAddress* parameter must, therefore, be zero.

Calls to HvAssertVirtualInterrupt will fail if the interrupt asserted by a previous call has already been acknowledged by the processor. This acknowledgement must first be cleared by calling HvClearVirtualInterrupt. This is especially useful when implementing an external interrupt controller, such as the 8259 PIC. It prevents HvAssertVirtualInterrupt from overwriting the previous acknowledgement, which may need to be reported through the external interrupt controller.

Input Parameters

DestinationPartition specifies the partition.

InterruptControl specifies the type of the virtual interrupt that should be asserted, its destination mode and whether the virtual interrupt is edge or level triggered.

DestinationAddress specifies the destination virtual processor(s). In case of physical destination mode, the destination address specifies the physical APIC ID of the target virtual processor. In case of logical destination mode, the destination address specifies the logical APIC ID of the set of target virtual processors. This value must be zero for external interrupt delivery mode where the interrupt request is always sent to the boot processor.

RequestedVector specifies the interrupt vector. This value is used only for fixed, lowest-priority, external, and SIPI interrupt types. In all other cases, a vector of zero must be specified.

TargetVtl specifies the VTL to be targeted for this call.

Output Parameters

None.

Restrictions

- The partition specified by *DestinationPartition* must be in the "active" state.
- The caller must be the parent of the partition specified by *DestinationPartition*.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_VP_INDEX	The virtual processor selected by the *DestinationAddress* parameter is not valid.
	For interrupts of type HvX64InterruptTypeExtInt, the DestinationAddress was non-zero.
HV_STATUS_INVALID_PARAMETER	One or more fields of the specified interrupt control are invalid or reserved bits within the interrupt control are set.
	The specified destination address is invalid or is non-zero for an external interrupt type.

Status code	Error condition
	The specified vector is not within a valid range (0 to 255 inclusive or HV_INTERRUPT_VECTOR_NONE).
	A non-zero vector is specified with an interrupt type that is not fixed, lowest-priority, external, or SIPI.
HV_STATUS_ACKNOWLEDGED	An external interrupt cannot be asserted because a previously-asserted external interrupt was acknowledged by the virtual processor and has not yet been cleared.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INVALID_VTL_STATE	The VTL state conflicts with the requested VTL count property change.
HV_STATUS_OPERATION_DENIED	The operation could not be performed.

10.5.2 **HvSendSyntheticClusterIpi**

This hypercall sends a virtual **fixed** interrupt to the specified virtual processor set. It does not support NMIs.

Wrapper Interface

```
HV_STATUS
HvSendSyntheticClusterIpi(
      __in  UINT32          Vector;
      __in  HV_INPUT_VTL    TargetVtl;
      __in  UINT64          ProcessorMask;
      );
```

Native Interface

HvSendSyntheticClusterIpi	
	Call Code = 0x000b
➡ Input Parameters	

0	Vector (4 bytes)	TargetVtl (1 byte)	Rsvd (3 bytes)
8	ProcessorMask (8 bytes)		

Input Parameters

*Vector s*pecifies the vector asserted. Must be between >= 0x10 and <= 0xFF.

TargetVtl specifies the VTL to target.

ProcessorMask specifies a mask consisting of HV_VP_INDEX, representing which VPs to target.

Output Parameters

None.

Restrictions

- This hypercall does not support NMI's (non-maskable interrupts)

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	One or more fields of the specified interrupt control are invalid or reserved bits within the interrupt control are set.

10.5.3 **HvSendSyntheticClusterIpiEx**

This hypercall sends a virtual **fixed** interrupt to the specified virtual processor set. It does not support NMIs. This version differs from HvSendSyntheticClusterIpi in that a variable sized VP set can be specified.

The following checks should be used to infer the availability of this hypercall:

1. ExProcessorMasks must be indicated via CPUID leaf 0x40000004 (see 2.4.5)

Wrapper Interface

```
HV_STATUS
HvSendSyntheticClusterIpiEx(
      __in  UINT32            Vector;
      __in  HV_INPUT_VTL      TargetVtl;
      __in  HV_VP_SET         ProcessorSet;
      );
```

Native Interface

HvSendSyntheticClusterIpiEx	
	Call Code = 0x0015

➡ Input Parameters			
0	Vector (4 bytes)	TargetVtl (1 byte)	Rsvd (3 bytes)
8	ProcessorSet (variably sized)		

Input Parameters

Vector specifies the vector asserted. Must be between >= 0x10 and <= 0xFF.

TargetVtl specifies the VTL to target.

ProcessorSet specifies a set consisting of HV_VP_SET representing which VPs to target.

Output Parameters

None.

Restrictions

- This hypercall does not support NMI's (non-maskable interrupts)

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	One or more fields of the specified interrupt control are invalid or reserved bits within the interrupt control are set.

10.5.4 HvRetargetDeviceInterrupt

This hypercall retargets a device interrupt, which may be useful for rebalancing IRQs within a guest.

Wrapper Interface

```
HV_STATUS
HvRetargetDeviceInterrupt(
      __in  HV_PARTITION_ID               PartitionId,
      __in  UINT64                        DeviceId,
      __in  HV_INTERRUPT_ENTRY            InterruptEntry,
      __in  UINT64                        Reserved,
      __in  HV_DEVICE_INTERRUPT_TARGET    InterruptTarget
      );
```

Native Interface

HvRetargetDeviceInterrupt	
	Call Code = 0x007e

➡ Input Parameters	
0	PartitionId (8 bytes)
8	DeviceId (8 bytes)
16	InterruptEntry (16 bytes)
24	
32	Reserved
40	InterruptTarget (16 bytes)
48	

Input Parameters

PartitionId: must be HV_PARTITION_SELF (-1)

DeviceId: supplies the unique (within a guest) logical device ID that is assigned by the host.

InterruptEntry: supplies the MSI address and data that identifies the interrupt (see 10.4.6).

InterruptTarget: specifies the new virtual interrupt target (see 10.4.7).

Output Parameters

None.

Restrictions

- Virtual processor indices specified by the processor mask must exist at the time of calling. Specifying the special “all processors” type is invalid for this hypercall.
- Reserved fields must be 0.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller did not possess sufficient access rights to perform the requested operation.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.
HV_STATUS_FEATURE_UNAVAILABLE	A hypervisor feature is not available to the caller.

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	One or more fields of the input structure are invalid or reserved bits are set.
HV_STATUS_INVALID_DEVICE_ID	The supplied device ID is invalid.
HV_STATUS_OPERATION_DENIED	The operation could not be performed. (The actual cause depends on the operation.)

11 Inter-Partition Communication

11.1 Overview

The hypervisor provides two simple mechanisms for one partition to communicate with another: messages and events. In both cases, notification is signaled by using the SynIC (synthetic interrupt controller).

11.2 SynIC Messages

The hypervisor provides a simple inter-partition communication facility that allows one partition to send a parameterized message to another partition. (Because the message is sent asynchronously, it is said to be *posted*.) The destination partition may be notified of the arrival of this message through an interrupt.

11.3 Message Buffers

A *message buffer* is used internally to the hypervisor to store a message until it is delivered to the recipient. The hypervisor maintains several sets of message buffers.

11.3.1 Guest Message Buffers

The hypervisor maintains a set of guest message buffers for each port. These buffers are used for messages sent explicitly from one partition to another by a guest. When a port is created, the hypervisor will allocate sixteen (16) message buffers from the port owner's memory pool. These message buffers are returned to the memory pool when the port is deleted.

11.3.2 Timer Message Buffers

The hypervisor maintains four timer message buffers for each virtual processor (one per synthetic interrupt timer). They are allocated when a virtual processor is created.

11.3.3 Intercept Message Buffers

The hypervisor maintains one intercept message buffer for each virtual processor. It is used for intercepts. The intercept message buffer is allocated when the virtual processor is created.

11.3.4 Event Log Message Buffers

The hypervisor maintains one event log message buffer for each event log group. It is used to notify the root partition when one or more event log buffers are full.

11.3.5 Message Buffer Queues

For each partition and each virtual processor in the partition, the hypervisor maintains one queue of message buffers for each SINTx (synthetic interrupt source) in the virtual processor's SynIC. All message queues of a virtual processor are empty upon creation or reset of the virtual processor.

11.3.6 Reliability and Sequencing of Guest Message Buffers

Messages successfully posted by a guest have been queued for delivery by the hypervisor. Actual delivery and reception by the target partition is dependent upon its correct operation. Partitions may disable delivery of messages to particular virtual processors by either disabling its SynIC or disabling the SIMP.

Breaking a connection will not affect undelivered (queued) messages. Deletion of the target port will always free all of the port's message buffers, whether they are available or contain undelivered (queued) messages.

Messages arrive in the order in which they have been successfully posted. If the receiving port is associated with a specific virtual processor, then messages will arrive in the same order in which they were posted. If the receiving port is associated with HV_ANY_VP, then messages are not guaranteed to arrive in any particular order.

11.4 Messages

When a message is sent, the hypervisor selects a free message buffer. The set of available message buffers depends on the event that triggered the sending of the message.

The hypervisor marks the message buffer "in use" and fills in the message header with the message type, payload size, and information about the sender. Finally, it fills in the message payload. The contents of the payload depend on the event that triggered the message. This document specifies the payloads of all messages generated by the hypervisor. The payload for messages sent by calling HvPostMessage must be defined by the caller.

The hypervisor then appends the message buffer to a receiving message queue. The receiving message queue depends on the event that triggered the sending of the message. For all message types, SINTx is either implicit (in the case of intercept messages), explicit (in the case of timer messages) or specified by a port ID (in the case of guest messages). The target virtual processor is either explicitly specified or chosen by the hypervisor when the message is enqueued. Virtual processors whose SynIC or SIM page (see section 11.9) is disabled will not be considered as potential targets. If no targets are available, the hypervisor terminates the operation and returns an error to the caller.

The hypervisor then determines whether the specified SINTx message slot within the SIM page for the target virtual processor is empty. (See section 11.9 for a description of the SIM page.) If the message type in the message slot is equal to HvMessageTypeNone (that is, zero), the message slot is assumed to be empty. In this case, the hypervisor dequeues the message buffer and copies its contents to the message slot within the SIM page. The hypervisor may copy only the number of payload bytes associated with the message. The hypervisor also attempts to generate an edge-triggered interrupt for the specified SINTx. If the APIC is software disabled or the SINTx is masked, the interrupt is lost. The arrival of this interrupt notifies the guest that a new message has arrived. If the SIM page is disabled or the message slot within the SIM page is not empty, the message remains queued, and no interrupt is generated.

As with any fixed-priority interrupt, the interrupt is not acknowledged by the virtual processor until the PPR (process priority register) is less than the vector specified in the SINTx register and interrupts are not masked by the virtual processor (rFLAGS[IF] is set to 1).

Multiple message buffers with the same SINTx can be queued to a virtual processor. In this case, the hypervisor will deliver the first message (that is, write it to the SIM page) and leave the others queued until one of three events occur:

- Another message buffer is queued.
- The guest indicates the "end of interrupt" by writing to the APIC's EOI register.
- The guest indicates the "end of message" by writing to the SynIC's EOM register.

In all three cases, the hypervisor will scan one or more message buffer queues and attempt to deliver additional messages. The hypervisor also attempts to generate an edge-triggered interrupt, indicating that a new message has arrived.

If a queued message cannot be delivered because the corresponding SIM entry is still in use, the hypervisor will attempt to deliver it again after an unspecified time (typically on the order of milliseconds). To avoid this potential latency, software should mark the SIM entry as unused before indicating an EOI or EOM.

11.4.1 **Recommended Message Handling**

The SynIC message delivery mechanism is designed to accommodate efficient delivery and receipt of messages within a target partition. It is recommended that the message handling ISR (interrupt service routine) within the target partition perform the following steps:

- Examine the message that was deposited into the SIM message slot.
- Copy the contents of the message to another location and set the message type within the message slot to HvMessageTypeNone.
- Indicate the end of interrupt for the vector by writing to the APIC's EOI register.
- Perform any actions implied by the message.

11.4.2 **Message Sources**

The classes of events that can trigger the sending of a message are as follows:

- **Intercepts:** Any intercept in a virtual processor will cause a message to be sent. The message buffer used is the intercept message buffer of the virtual processor that caused the intercept. The receiving message queue belongs to SINT0 of a virtual processor that the hypervisor selects non-deterministically from among the virtual processors of the parent partition. The message payload describes the event that caused the intercept. If the intercept message buffer is already queued when an intercept occurs, it is removed from the queue, overwritten, and placed back on the queue. This should occur only if the software running in the parent partition clears the "suspended for intercept" register before receiving the intercept message. This situation is considered a programming error.
- **Timers:** The timer mechanisms defined in chapter 12 will cause messages to be sent. Associated with each virtual processor are four dedicated timer message buffers, one for each timer. The receiving message queue belongs to SINTx of the virtual processor whose timer triggered the sending of the message.
- **Guest messages:** The hypervisor supports message passing as an inter-partition communication mechanism between guests. The interfaces defined in this section allow one guest to send messages to another guest. The message buffers used for messages of this class are taken from the receiver's per-port pool of guest message buffers.
- **Event log buffers:** The hypervisor will send a message when an event log buffer has been filled.

11.5 SynIC Event Flags

In addition to messages, the SynIC supports a second type of cross-partition notification mechanism called *event flags*. Event flags may be set explicitly using the HvSignalEvent hypercall or implicitly by the hypervisor as a consequence of the monitored notification facility.

11.5.1 **Event Flag Delivery**

When a partition calls HvSignalEvent, it specifies an event flag number. The hypervisor responds by atomically setting a bit within the receiving virtual processor's SIEF page. (See section 11.9 for a detailed description of the SIEF page.) Virtual processors whose SynIC or SIEF page is disabled will not be considered as potential targets. If no targets are available, the hypervisor terminates the operation and returns an error to the caller.

If the event flag was previously cleared, the hypervisor attempts to notify the receiving partition that the flag is now set by generating an edge-triggered interrupt. The target virtual processor, along with the target SINTx, is specified as part of a port's creation. (See the following for information about ports.) If the SINTx is masked, HvSignalEvent returns HV_STATUS_INVALID_SYNIC_STATE.

As with any fixed-priority external interrupt, the interrupt is not acknowledged by the virtual processor until the process priority register (PPR) is less than the vector specified in the SINTx register and interrupts are not masked by the virtual processor (rFLAGS[IF] is set to 1).

11.5.2 **Recommended Event Flag Handling**

It is recommended that the event flag interrupt service routine (ISR) within the target partition perform the following steps:

- Examine the event flags and determine which ones, if any, are set.
- Clear one or more event flags by using a locked (atomic) operation such as LOCK AND or LOCK CMPXCHG.
- Indicate the end of interrupt for the vector by writing to the APIC's EOI register.
- Perform any actions implied by the event flags that were set.

11.5.3 **Event Flags versus Messages**

Event flags are lighter-weight than messages and are therefore lower overhead. Furthermore, event flags do not require any buffer allocation or queuing within the hypervisor, so HvSignalEvent will never fail due to insufficient resources.

11.6 Ports and Connections

A message or event sent from one guest to another must be sent through a pre-allocated *connection.* A connection, in turn, must be associated with a destination *port*.

A port is allocated from the receiver's memory pool and specifies which virtual processor and SINTx to target. Event ports have a "base flag number" and "flag count" that allow the caller to specify a range of valid event flags for that port.

Connections are allocated from the sender's memory pool. When a connection is created, it must be associated with a valid port. This binding creates a simple, one-way communication channel. If a port is subsequently deleted, its connection, while it remains, becomes useless.

11.7 Monitored Notifications

The monitored notification facility (MNF) introduces the concept of shared triggers between two communicating partitions. MNF uses a port (in the recipient partition) and a connection (in the originating partition) to establish a hypervisor-monitored, unidirectional notification channel. A monitor port-and-connection pair alone isn't enough to form the said notification channel. It needs in addition to be associated with an event connection through the monitored notification parameters in the monitored notification page

When the channel is created, a monitored notification is established in an overlay page that includes the following:

- A trigger,
- A latency hint
- A set of input parameters appropriate for the HvSignalEvent hypercall.

After the monitor page is established, the hypervisor periodically examines the trigger at a rate subject to the latency hint to determine if a notification is warranted. If so, the hypervisor invokes the HvSignalEvent hypercall internally on behalf of the originating guest. The behavior is the same as if the originating guest had invoked the HvSignalEvent directly.

11.7.1 **Monitored Notification Trigger**

The trigger can be directly accessed by guests without hypervisor intervention. It is set or cleared by the inter-partition communication code running in the communicating guests. The trigger must be placed in memory that is shared by the two communicating partitions and the hypervisor.

11.7.2 **Monitored Notification Latency Hint**

The latency hint specifies an approximate wait period between hypervisor examinations of the trigger. It is expressed in 100 nanosecond units. The hypervisor can override the specified latency value if making it somewhat smaller or larger is more efficient. The hypervisor can also override the specified latency value if it exceeds minimum or maximum values.

11.7.3 **Monitored Notification Parameters**

Each MNF trigger is defined by a set of input parameters compatible with those accepted by an HvSignalEvent hypercall. These parameters include an event flag number and a connection ID. If the internal invocation of the HvSignalEvent hypercall fails, the error is discarded and the invocation is treated as a NOP.

11.7.4 **Monitored Notification Page**

Monitored notifications are collected into monitor overlay pages that can be created or deleted only from a parent partition. The parent partition creates a monitor-page port in the recipient and specifies the GPA of the recipient's associated monitor page. The parent subsequently creates a connection to that the monitor page port in the originator and specifies the GPA of the originator's associated monitor page. While each of these two GPAs is partition-specific, the underlying physical page is a common page that is managed by the hypervisor. Changes to the page are visible from both partitions as well as the hypervisor.

11.8 SynIC MSRs

In addition to the memory-mapped registers defined for a local APIC, the following model-specific registers (MSRs) are defined in the SynIC. Each virtual processor has its own copy of these registers, so they can be programmed independently.

MSR Address	Register Name	Function
0x40000080	SCONTROL	SynIC Control
0x40000081	SVERSION	SynIC Version
0x40000082	SIEFP	Interrupt Event Flags Page
0x40000083	SIMP	Interrupt Message Page
0x40000084	EOM	End of message
0x40000090	SINT0	Interrupt source 0 (hypervisor)
0x40000091	SINT1	Interrupt source 1
0x40000092	SINT2	Interrupt source 2
0x40000093	SINT3	Interrupt source 3
0x40000094	SINT4	Interrupt source 4
0x40000095	SINT5	Interrupt source 5
0x40000096	SINT6	Interrupt source 6
0x40000097	SINT7	Interrupt source 7
0x40000098	SINT8	Interrupt source 8
0x40000099	SINT9	Interrupt source 9
0x4000009A	SINT10	Interrupt source 10
0x4000009B	SINT11	Interrupt source 11
0x4000009C	SINT12	Interrupt source 12
0x4000009D	SINT13	Interrupt source 13
0x4000009E	SINT14	Interrupt source 14
0x4000009F	SINT15	Interrupt source 15

11.8.1 **SCONTROL Register**

63:1	0
RsvdP	Enable

This register is used to control SynIC behavior of the virtual processor.

Bits	Description	Attributes
63:1	RsvdP (value must be preserved)	Read/write
0	Enable When set, this virtual processor will allow message queuing and event flag notifications to be posted to its SynIC. When clear, message queuing and event flag notifications cannot be directed to this virtual processor.	Read/write

At virtual processor creation time and upon processor reset, the value of this SCONTROL (SynIC control register) is 0x0000000000000000. Thus, message queuing and event flag notifications will be disabled.

11.8.2 **SVERSION Register**

63:32	31:0
Rsvd	SynIC Version (0x00000001)

This is a read-only register, and it returns the version number of the SynIC. For the first version of the hypervisor, the value is 0x00000001. Attempts to write to this register result in a #GP fault.

11.8.3 **SIEFP Register**

63:12	11:1	0
SIEFP Base Address	RsvdP	Enable

Bits	Description	Attributes
63:12	Base address (in GPA space) of SIEFP (low 12 bits assumed to be zero)	Read/write
11:1	RsvdP (value should be preserved)	Read/write
0	SIEFP enable	Read/write

At virtual processor creation time and upon processor reset, the value of this SIEFP (synthetic interrupt event flags page) register is 0x0000000000000000. Thus, the SIEFP is disabled by default. The guest must enable it by setting bit 0. If the specified base address is beyond the end of the partition's GPA space, the SIEFP page will not be accessible to the guest. When modifying the register, guests should preserve the value of the reserved bits (1 through 11) for future compatibility.

11.8.4 **SIMP Register**

63:12	11:1	0
SIMP Base Address	RsvdP	Enable

Bits	Description	Attributes
63:12	Base address (in GPA space) of SIMP (low 12 bits assumed to be zero)	Read/write
11:1	RsvdP (value should be preserved)	Read/write
0	SIMP enable	Read/write

At virtual processor creation time and upon processor reset, the value of this SIMP (synthetic interrupt message page) register is 0x0000000000000000. Thus, the SIMP is disabled by default. The guest must enable it by setting bit 0. If the specified base address is beyond the end of the partition's GPA space, the SIMP page will not be accessible to the guest. When modifying the register, guests should preserve the value of the reserved bits (1 through 11) for future compatibility.

11.8.5 **SINTx Registers**

63:19	18	17	16	15:8	7:0
RsvdP	Polling	AutoEOI	Mask	RsvdP	Vector

Bits	Description	Attributes
63:19	RsvdP (value should be preserved)	Read/write
18	Polling	Read/write
17	AutoEOI Set if an implicit EOI should be performed upon interrupt delivery	Read/write
16	Set if the SINT is masked	Read/write
15:8	RsvdP (value should be preserved)	Read/write
7:0	Vector	Read/write

At virtual processor creation time, the default value of all SINTx (synthetic interrupt source) registers is 0x0000000000010000. Thus, all synthetic interrupt sources are masked by default. The guest must unmask them by programming an appropriate vector and clearing bit 16.

Setting the polling bit will have the effect of unmasking an interrupt source, except that an actual interrupt is not generated.

The AutoEOI flag indicates that an implicit EOI should be performed by the hypervisor when an interrupt is delivered to the virtual processor. In addition, the hypervisor will automatically clear the corresponding flag in the "in-service register" (ISR) of the virtual APIC. If the guest enables this behavior, then it must not perform an EOI in its interrupt service routine.

The AutoEOI flag can be turned on at any time, though the guest must perform an explicit EOI on an in-flight interrupt The timing consideration makes it difficult to know whether a particular interrupt needs EOI or not, so it is recommended that once SINT is unmasked, its settings are not changed.

Likewise, the AutoEOI flag can be turned off at any time, though the same concerns about in-flight interrupts apply

Valid values for v*ector* are 16-255 inclusive. Specifying an invalid vector number results in #GP.

11.8.6 **EOM Register**

63:0
RsvdZ

Bits	Description	Attributes
63:0	RsvdZ (value should be set to zero)	Write-only trigger

A write to the end of message (EOM) register by the guest causes the hypervisor to scan the internal message buffer queue(s) associated with the virtual processor. If a message buffer queue contains a queued message buffer, the hypervisor attempts to deliver the message. Message delivery succeeds if the SIM page is enabled and the message slot corresponding to the SINTx is empty (that is, the message type in the header is set to HvMessageTypeNone). If a message is successfully delivered, its corresponding internal message buffer is dequeued and marked free. If the corresponding SINTx is not masked, an edge-triggered interrupt is delivered (that is, the corresponding bit in the IRR is set).

This register can be used by guests to "poll" for messages. It can also be used as a way to drain the message queue for a SINTx that has been disabled (that is, masked).

If the message queues are all empty, a write to the EOM register is a no-op.

Reads from the EOM register always returns zeros.

11.9 SIM and SIEF Pages

The SynIC defines two pages that extend the functionality of a traditional APIC. The pages for these two addresses are specified by the SIEFP register and the SIMP register (see earlier in this specification for these register formats).

The SIEF and SIM pages are implemented as GPA overlay pages. For a description of overlay pages, see 5.2.1.

The addresses of the SIEF and SIM pages should be unique for each virtual processor. Programming these pages to overlap other instances of the SIEF or SIM pages or any other overlay page (for example, the hypercall page) will result in undefined behavior.

The hypervisor may implement the SIEF and SIM pages so that a SIEF or SIM instance associated with a virtual processor is not accessible to other virtual processors. In such implementations, an access by one virtual processor to another virtual processor's SIEF or SIM page will result in a #MC fault. It is highly recommended that guests avoid performing such accesses.

Read and write accesses by a virtual processor to the SIEF and SIM pages behave like read and write accesses to RAM. However, the hypervisor's SynIC implementation also writes to these pages in response to certain events.

Upon virtual processor creation and reset, the SIEF and SIM pages are cleared to zero.

The SIEF page consists of a 16-element array of 256-byte event flags (see the following for an explanation of event flags). Each array element corresponds to a single synthetic interrupt source (SINTx).

The SIM page consists of a 16-element array of 256-byte messages (see the following HV_MESSAGE data structure). Each array element (also known as a *message slot*) corresponds to a single synthetic interrupt source (SINTx). A message slot is said to be "empty" if the message type of the message in the slot is equal to HvMessageTypeNone.

11.10 Inter-Partition Communication Data Types

11.10.1 **Synthetic Interrupt Sources**

The SynIC supports 16 synthetic interrupt sources.

```
#define HV_SYNIC_SINT_COUNT   16
```

```
typedef UINT32 HV_SYNIC_SINT_INDEX;
```

11.10.2 **SynIC Message Types**

```
SynIC messages encode the message type as a 32-bit number.
typedef enum
{
      HvMessageTypeNone = 0x00000000,

      // Memory access messages
      HvMessageTypeUnmappedGpa      = 0x80000000,
      HvMessageTypeGpaIntercept     = 0x80000001,

      // Timer notifications
      HvMessageTimerExpired   = 0x80000010,

      // Error messages
      HvMessageTypeInvalidVpRegisterValue = 0x80000020,
      HvMessageTypeUnrecoverableException = 0x80000021,
      HvMessageTypeUnsupportedFeature     = 0x80000022,
      HvMessageTypeTlbPageSizeMismatch    = 0x80000023,

      // Trace buffer messages
      HvMessageTypeEventLogBuffersComplete      = 0x80000040,

      // Hypercall intercept.
      HvMessageTypeHypercallIntercept = 0x80000050,

      // Platform-specific processor intercept messages
      HvMessageTypeX64IoPortIntercept     = 0x80010000,
      HvMessageTypeMsrIntercept           = 0x80010001,
      HvMessageTypeX64CpuidIntercept      = 0x80010002,
      HvMessageTypeExceptionIntercept     = 0x80010003,
      HvMessageTypeX64ApicEoi             = 0x80010004,
      HvMessageTypeX64LegacyFpError       = 0x80010005,
      HvMessageTypeRegisterIntercept      = 0x80010006,
} HV_MESSAGE_TYPE;


#define HV_MESSAGE_TYPE_HYPERVISOR_MASK   0x80000000
```

Any message type that has the high bit set is reserved for use by the hypervisor. Guest-initiated messages cannot send messages with a hypervisor message type.

11.10.3 **SynIC Message Flags**

7:1	0
RsvdZ	MessagePending

Bits	Description	Meaning
7:1	RsvdP (value should be set to zero)	Reserved
0	MessagePending	One or more messages are pending in the message queue

The *MessagePending* flag indicates whether or not there are any messages pending in the message queue of the synthetic interrupt source. If there are, then an "end of message" must be performed by the guest after emptying the message slot. This allows for opportunistic writes to the EOM MSR (only when required). Note that this flag may be set by the hypervisor upon message delivery or at any time afterwards. The flag should be tested after the message slot has been emptied and if set, then there are one or more pending messages and the "end of message" should be performed.

```
typedef struct
{
      UINT8 MessagePending:1;
      UINT8 Reserved:7;
} HV_MESSAGE_FLAGS;
```

11.10.4 **SynIC Message Format**

SynIC messages are of fixed size composed of a message header (which includes the message type and information about where the message originated) followed by the payload. Messages that are sent in response to HvPostMessage contain the port ID. Intercept messages contain the partition ID of the partition whose virtual processor generated the intercept. Timer intercepts do not have an origination ID (that is, the specified ID is zero).

```
#define HV_MESSAGE_SIZE       256
#define HV_MESSAGE_MAX_PAYLOAD_BYTE_COUNT 240
#define HV_MESSAGE_MAX_PAYLOAD_QWORD_COUNT      30


typedef struct
{
      HV_MESSAGE_TYPE   MessageType;
      UINT16      Reserved;
      HV_MESSAGE_FLAGS  MessageFlags;
      UINT8 PayloadSize;
      union
      {
        UINT64          OriginationId;
            HV_PARTITION_ID         Sender;
            HV_PORT_ID        Port;
      };
} HV_MESSAGE_HEADER;

typedef struct
{
      HV_MESSAGE_HEADER Header;
      UINT64      Payload[HV_MESSAGE_MAX_PAYLOAD_QWORD_COUNT];
} HV_MESSAGE;
```

11.10.5 **SynIC Event Flags**

SynIC event flags are fixed-size bitwise arrays. They are numbered such that the first byte of the array contains flags 0 through 7 (0 being the least significant bit) and the second byte of the array contains flags 8 through 15 (8 being the least significant bit), and so on.

```
#define HV_EVENT_FLAGS_COUNT               (256 * 8)
#define HV_EVENT_FLAGS_BYTE_COUNT   256


typedef struct
{
      UINT8 Flags[HV_EVENT_FLAGS_BYTE_COUNT];
} HV_SYNIC_EVENT_FLAGS;
```

11.10.6 **Ports**

Destination ports are identified by 32-bit IDs. The high 8 bits of the ID are reserved and must be zero. All port IDs are unique within a partition.

```
typedef union
{
      UINT32 AsUint32;
      struct
      {
          UINT32 Id:24;
          UINT32 Reserved:8;
      };
} HV_PORT_ID;
```

Three types of ports are supported: message ports, event ports, and monitor ports. Message ports are valid for use with the HvPostMessage hypercall. Event ports are valid for use with the HvSignalEvent hypercall. Monitor ports are valid for use with monitor pages that are monitored by the hypervisor and result in HvSignalEvent-based notifications when appropriate.

```
enum HV_PORT_TYPE
{
      HvPortTypeMessage = 1,
      HvPortTypeEvent   = 2,
      HvPortTypeMonitor = 3
};
```

When a port is created, the following information is specified.

```
typedef struct
{
      HV_PORT_TYPE                  PortType;
      UINT32            ReservedZ;

      union
      {
          struct
        {
              HV_SYNIC_SINT_INDEX   TargetSint;
              HV_VP_INDEX           TargetVp;
              UINT64                ReservedZ;
        } MessagePortInfo;

        struct
        {
              HV_SYNIC_SINT_INDEX   TargetSint;
              HV_VP_INDEX           TargetVp;
              UINT16                BaseFlagNumber;
              UINT16                FlagCount;
              UINT32                ReservedZ;
        } EventPortInfo;

        struct
          {
              HV_GPA                MonitorAddress;
              UINT64                ReservedZ;
          } MonitorPortInfo;
      };
} HV_PORT_INFO;
```

11.10.7 **Connections**

Connections are identified by 32-bit IDs. The high 8 bits are reserved and must be zero. All connection IDs are unique within a partition.

```
typedef union
{
      UINT32 AsUint32;
      struct
      {
          UINT32 Id:24;
          UINT32 Reserved:8;
      };
} HV_CONNECTION_ID;
```

The hypervisor does not ascribe special meaning to any connection IDs.

11.10.8 **Connection Information**

The following structure contains the information that must be specified when creating a connection:

```
typedef struct
{
      HV_PORT_TYPE            PortType;
      UINT32                  ReservedZ;

      union
      {
            struct
            {
                  UINT64      ReservedZ;
            } MessageConnectionInfo;

            struct
            {
                  UINT64      ReservedZ;
            } EventConnectionInfo;

            struct
            {
                  HV_GPA      MonitorAddress;
            } MonitorConnectionInfo;
      };
} HV_CONNECTION_INFO, *PHV_CONNECTION_INFO;
```

11.10.8.1 **Monitored Notification Trigger Group**

The monitored notification triggers group structure defines 32 triggers per group. The structure has the following format:

```
typedef struct
{
      UINT64            AsUINT64;
      struct
      {
            UINT32      Pending;
            UINT32      Armed;
      };
} HV_MONITOR_TRIGGER_GROUP, *PHV_MONITOR_TRIGGER_GROUP;
```

The 32 triggers are represented by two related arrays of bits: *Pending* and *Armed*. Setting a trigger bit to 1 in the *Pending* array indicates to the hypervisor that the related notification should eventually generate a signal event. The corresponding bit in the *Armed* array should be set to 0 whenever the matching *Pending* bit is modified. The *Armed* bit is used to ensure that a notification is deferred by at least the latency specified for the notification. *Both of these bits must be updated atomically.*

11.10.8.2 **Monitored Notification Parameters**

Each trigger has a set of associated notification parameters that are used by the hypervisor as inputs to the implicit HvSignalEvent hypercall that the hypervisor invokes when appropriate. The parameter structure has the following format:

```
typedef struct
{
      HV_CONNECTION_ID        ConnectionId;
      UINT16                  FlagNumber;
      UINT16                  ReservedZ;
} HV_MONITOR_PARAMETER, *PHV_MONITOR_PARAMETER;
```

When the hypervisor detects that a monitored notification is pending, it signals the event by making an internal call to the HvSignalEvent hypercall and passing it the *ConnectionID* and *FlagNumber* members. If signaling an event causes an error, the error is discarded; that is, the internal HvSignalEvent call becomes a NOP.

11.10.9 **Monitored Notification Page**

Monitored notifications are defined in a *MNF overlay page*, which supports four sets of monitored notification trigger groups. Each individual 32-bit group can be enabled independently using the following structure:

```
typedef union
{
      UINT32                        AsUINT32;

      struct
      {
            UINT32                  GroupEnable:4;
            UINT32                  ReservedZ:27;
      };
} HV_MONITOR_TRIGGER_STATE, *PHV_MONITOR_TRIGGER_STATE;
```

GroupEnable is described below.

The MNF overlay page has the following format:

```
typedef struct
{
      HV_MONITOR_TRIGGER_STATE      TriggerState;
      UINT32                        Reserved1;

      HV_MONITOR_TRIGGER_GROUP      TriggerGroup[4];
      UINT8                         Reserved2[536];

      UINT16                        Latency[4][32];
      UINT8                         Reserved3[256];

      HV_MONITOR_PARAMETER          Parameter[4][32];

      UINT8                         Reserved4[1984];
} HV_MONITOR_PAGE, *PHV_MONITOR_PAGE;

typedef volatile HV_MONITOR_PAGE *PVHV_MONITOR_PAGE;
```

TriggerState contains the *GroupEnable* flags. The *GroupEnable* flags are an array of 4 bits, each associated with a trigger group. If *GroupEnable[n]* is set to one, the corresponding *TriggerGroup[n]* is enabled. Although the *GroupEnable* flags can be changed at any time, they are intended to be semi-static and are typically used to drain pending notifications during a save or restore process. The hypervisor inspects the group enable flags at varying rates. If they are all disabled (set to zero), the hypervisor might significantly reduce its inspection rate. The hypervisor inspects all of the enabled monitored notifications approximately at the lowest latency value specified for the monitors of each group.

TriggerGroup is an array of four trigger group structures. For details, see section 11.10.8.1.

Latency is a hint; suggesting how often the hypervisor should inspect the monitored notifications. The hypervisor might adjust it to be smaller or larger than this value if doing so is either more efficient or to conform to implementation-specific limitations. Latency is specified in 100-nanosecond units.

Parameter is an array of notification parameters, one per trigger. The hypervisor can monitor up to 128 notifications in groups of 32. For details, see section 11.10.8.2 .

The *Reservedn* bits are reserved for use by the hypervisor. Changing their value is boundedly undefined.

11.11 Inter-Partition Communication Interfaces

11.11.1 **HvPostMessage**

The HvPostMessage hypercall attempts to post (that is, send asynchronously) a message to the specified connection, which has an associated destination port.

Wrapper Interface

```
HV_STATUS
HvPostMessage(
      __in HV_CONNECTION_ID   ConnectionId,
      __in HV_MESSAGE_TYPE    MessageType,
      __in UINT32 PayloadSize,
      __in_ecount(PayloadSize)
           PCVOID Message
      );
```

Native Interface

HvPostMessage		
	Call Code = 0x005C	
➡ Input Parameters		
0	ConnectionId (4 bytes)	Padding (4 bytes)
8	MessageType (4 bytes)	PayloadSize (4 bytes)
16	Message[0] (8 bytes)	
⋮	⋮	
248	Message[29] (8 bytes)	

Description

If the message is successfully posted, then it will be queued for delivery to a virtual processor within the partition associated with the port.

For details about message delivery, see section 11.4.

Input Parameters

ConnectionId specifies the ID of the connection created by calling HvConnectPort.

MessageType specifies the message type that will appear within the message header. The caller can specify any 32-bit message type whose most significant bit is cleared, with the exception of zero. Message types with the high bit set are reserved for use by the hypervisor.

PayloadSize specifies the number of bytes that are included in the message.

Message specifies the payload of the message—up to 240 bytes total. Only the first *n* bytes are actually sent to the destination partition, where *n* is provided in the *PayloadSize* parameter.

Output Parameters

None.

Restrictions

- The partition that is the target of the connection must be in the “active” state.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller’s partition does not possess the *PostMessages* privilege.
HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is invalid.
HV_STATUS_INVALID_PORT_ID	The port associated with the specified connection has been deleted.
	The port associated with the specified connection belongs to a partition that is not in the “active” state.
	The port associated with the specified connection is not a "message" type port.
HV_STATUS_INVALID_PARAMETER	The most significant bit of the specified message type is set.
	The *MessageType* parameter specifies a value of zero.
	The specified payload size exceeds 240 bytes.
HV_STATUS_INSUFFICIENT_BUFFERS	The port has no available guest message buffers.
HV_STATUS_INVALID_VP_INDEX	The target VP no longer exists or there are no available VPs to which the message can be posted.
HV_STATUS_INVALID_SYNIC_STATE	The target VP’s SynIC is disabled and cannot accept posted messages. For ports targeted at HV_ANY_VP, this indicates that the SynIC of all of the partition’s VPs are disabled.

Status code	Error condition
	The target VP's SIM page is disabled. For ports targeted at HV_ANY_VP, this indicates that the SIM page of all of the partition's VPs are disabled.

11.11.2 **HvSignalEvent**

The HvSignalEvent hypercall signals an event in a partition that owns the port associated with the specified connection.

Wrapper Interface

```
HV_STATUS
HvSignalEvent(
      __in HV_CONNECTION_ID   ConnectionId,
      __in UINT16 FlagNumber
      );
```

Native Interface

HvSignalEvent			
	Call Code = 0x005D		
➡ Input Parameter Header			
0	ConnectionId (4 bytes)	FlagNumber (2 bytes)	RsvdZ (2 bytes)

Description

The event is signaled by setting a bit within the SIEF page of one of the receive partition's virtual processors.

The caller specifies a relative flag number. The actual SIEF bit number is calculated by the hypervisor by adding the specified flag number to the base flag number associated with the port.

Input Parameters

ConnectionId specifies the ID of the connection.

FlagNumber specifies the relative index of the event flag that the caller wants to set within the target SIEF area. This number is relative to the base flag number associated with the port.

Output Parameters

None.

Restrictions

- The partition that is the target of the connection must be in the “active” state.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller’s partition does not possess the *SignalEvents* privilege.
HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is invalid.
HV_STATUS_INVALID_PORT_ID	The port associated with the specified connection has been deleted.
	The port associated with the specified connection belongs to a partition that is not in the “active” state.
	The port associated with the specified connection is not an "event" type port.
HV_STATUS_INVALID_PARAMETER	The specified flag number is greater than or equal to the port’s flag count.
HV_STATUS_INVALID_VP_INDEX	The target VP no longer exists or there are no available VPs to which the message can be posted.
HV_STATUS_INVALID_SYNIC_STATE	The target VP’s SynIC is disabled and cannot accept signaled events. For ports targeted at HV_ANY_VP, this indicates that the SynIC of all of the partition’s VPs are disabled.
	The target VP’s SIEF page is disabled. For ports targeted at HV_ANY_VP, this indicates that the SIEF page of all of the partition’s VPs are disabled.
	The target SINTx is masked.

12 Timers

12.1 Overview

12.1.1 Timer Services

The hypervisor provides simple timing services. These are based on a constant-rate reference time source (typically the ACPI timer on x64 systems).

The following timer services are provided:

- A per-partition reference time counter.
- Four synthetic timers per virtual processor. Each synthetic timer is a single-shot or periodic timer that delivers a message or asserts an interrupt when it expires.
- One virtual APIC timer per virtual processor.
- Two timer assists: an emulated periodic timer and a PM Timer assist.
- A partition reference time enlightenment, based on the host platform's support for an Invariant Time Stamp Counter (iTSC).

12.1.2 Reference Counter

The hypervisor maintains a per-partition reference time counter. It has the characteristic that successive accesses to it return strictly monotonically increasing (time) values as seen by any and all virtual processors of a partition. Furthermore, the reference counter is rate constant and unaffected by processor or bus speed transitions or deep processor power savings states. A partition's reference time counter is initialized to zero when the partition is created. The reference counter for all partitions count at the same rate, but at any time, their absolute values will typically differ because partitions will have different creation times.

The reference counter continues to count up as long as at least one virtual processor is not explicitly suspended.

12.1.3 Synthetic Timers

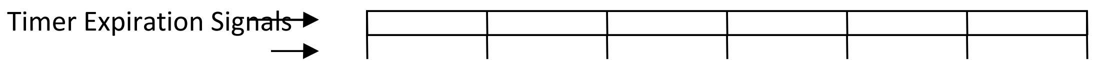
Synthetic timers provide a mechanism for generating an interrupt after some specified time in the future. Both one-shot and periodic timers are supported. A synthetic timer sends a message to a specified SynIC SINTx (synthetic interrupt source) upon expiration, or asserts an interrupt, depending on how it is configured.

The hypervisor guarantees that a timer expiration signal will never be delivered before the expiration time. The signal may arrive any time after the expiration time.

12.1.4 Periodic Timers

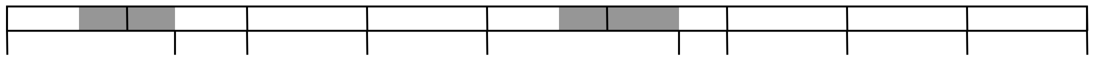
The hypervisor attempts to signal periodic timers on a regular basis.

For example, if a timer has a requested period of 1ms, here is the idealized schedule for timer expiration notifications:



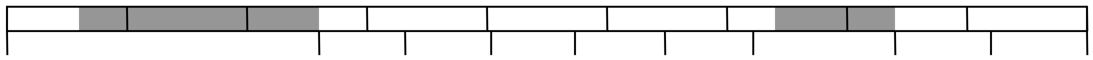
However, if the virtual processor used to signal the expiration is not available, some of the timer expirations may be delayed. A virtual processor may be unavailable because it is suspended (for example, during intercept handling) or because the hypervisor's scheduler decided that the virtual processor should not be scheduled on a logical processor (for example, because another virtual processor is using the logical processor or the virtual processor has exceeded its quota).

The shaded portions of the following diagram show periods of inactivity during which a periodic timer expiration signal could not be delivered. Consequently, the signal is deferred until the virtual processor becomes available.



If a virtual processor is unavailable for a sufficiently long period of time, a full timer period may be missed. In this case, the hypervisor uses one of two techniques. The first technique involves timer period modulation, in effect shortening the period until the timer "catches up".

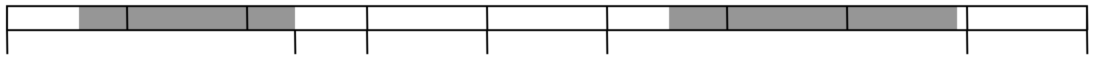
The following diagram shows the period modulation technique.



If a significant number of timer signals have been missed, the hypervisor may be unable to compensate by using period modulation. In this case, some timer expiration signals may be skipped completely.

For timers that are marked as *lazy*, the hypervisor uses a second technique for dealing with the situation in which a virtual processor is unavailable for a long period of time. In this case, the timer signal is deferred until this virtual processor is available. If it doesn't become available until shortly before the next timer is due to expire, it is skipped entirely.

The following diagram shows the lazy timer technique.



12.1.5 **Ordering of Timer Expirations**

Synthetic and virtualized timers generate interrupts at or near their designated expiration time. Due to hardware and other scheduling interactions, interrupts could potentially be delayed. No ordering may be assumed between any set of timers.

12.2 Direct Synthetic Timers

"Direct" synthetic timers assert an interrupt upon timer expiration instead of sending a message to a SynIc synthetic interrupt source (see Section 12.1.3 for more about synthetic timers).

A synthetic timer is set to "direct" mode by setting the "DirectMode" field of the synthetic timer configuration MSRs documented in section 12.5.1. The "ApicVector" field controls the interrupt vector that is asserted upon timer expiration.

12.3 Partition Reference Time Enlightenment

The partition reference time enlightenment presents a reference time source to a partition which does not require an intercept into the hypervisor. This enlightenment is available only when the underlying platform provides support of an invariant processor Time Stamp Counter (TSC), or iTSC. In such platforms, the processor TSC frequency remains constant irrespective of changes in the processor's clock frequency due to the use of power management states such as ACPI processor performance states, processor idle sleep states (ACPI C-states), etc.

The partition reference time enlightenment uses a virtual TSC value, an offset and a multiplier to enable a guest partition to compute the normalized reference time since partition creation, in 100nS units. The mechanism also allows a guest partition to atomically compute the reference time when the guest partition is migrated to a platform with a different TSC rate, and provides a fallback mechanism to support migration to platforms without the constant rate TSC feature.

This facility is not intended to be used a source of wall clock time, since the reference time computed using this facility will appear to stop during the time that a guest partition is saved until the subsequent restore.

12.4 Partition Reference Counter MSR

A partition's reference counter is accessed through a partition-wide MSR.

MSR Address	Register Name	Function
0x40000020	HV_X64_MSR_TIME_REF_COUNT	Time reference count (partition-wide)

12.4.1 **Reference Counter MSR**

63:0
Count

Bits	Description	Attributes
63:0	Count—Partition's reference counter value in 100 nanosecond units	Read-only

When a partition is created, the value of the TIME_REF_COUNT MSR is set to 0x0000000000000000. This value cannot be modified by a virtual processor. Any attempt to write to it results in a #GP fault.

12.5 Synthetic Timer MSRs

Synthetic timers are configured by using model-specific registers (MSRs) associated with each virtual processor. Each of the four synthetic timers has an associated pair of MSRs.

MSR address	Register name	Function
0x400000B0	HV_X64_MSR_STIMER0_CONFIG	Configuration register for synthetic timer 0
0x400000B1	HV_X64_MSR_STIMER0_COUNT	Expiration time or period for synthetic timer 0
0x400000B2	HV_X64_MSR_STIMER1_CONFIG	Configuration register for synthetic timer 1
0x400000B3	HV_X64_MSR_STIMER1_COUNT	Expiration time or period for synthetic timer 1
0x400000B4	HV_X64_MSR_STIMER2_CONFIG	Configuration register for synthetic timer 2
0x400000B5	HX_X64_MSR_STIMER2_COUNT	Expiration time or period for synthetic timer 2
0x400000B6	HV_X64_MSR_STIMER3_CONFIG	Configuration register for synthetic timer 3
0x400000B7	HV_X64_MSR_STIMER3_COUNT	Expiration time or period for synthetic timer 3

12.5.1 **Synthetic Timer Configuration Register**

63:20	19:16	15:13	12	11:4	3	2	1	0
RsvdZ	SINTx	RsvdZ	Direct Mode	APIC Vector	Auto Enable	Lazy	Periodic	Enable

Bits	Description	Attributes
63:20	RsvdZ (value should be set to zero)	Read/write
19:16	SINTx—synthetic interrupt source	Read/write
15:13	RsvdZ (value should be set to zero)	Read/write
12	DirectMode – assert an interrupt upon timer expiration. See section 12.2 for further information.	Read/write

Bits	Description	Attributes
11:4	ApicVector – controls the interrupt vector that is asserted upon timer expiration. See section 12.2 for further context.	Read/write
3	AutoEnable—set if writing the corresponding counter implicitly causes the counter to be enabled, cleared if not	Read/write
2	Lazy—set if timer is lazy, cleared if not	Read/write
1	Periodic—set if timer is periodic, cleared if one-shot	Read/write
0	Enable—set if timer is enabled	Read/write

When a virtual processor is created and reset, the value of all HV_X64_MSR_STIMERx_CONFIG (synthetic timer configuration) registers is set to 0x0000000000000000. Thus, all synthetic timers are disabled by default.

If *AutoEnable* is set, then writing a non-zero value to the corresponding count register will cause Enable to be set and activate the counter. Otherwise, Enable should be set after writing the corresponding count register in order to activate the counter. For information about the Count register, see the following section.

When a one-shot timer expires, it is automatically marked as disabled. Periodic timers remain enabled until explicitly disabled.

If a one-shot is enabled and the specified count is in the past, it will expire immediately.

It is not permitted to set the SINTx field to zero for an enabled timer. If attempted, the timer will be marked disabled (that is, bit 0 cleared) immediately.

Writing the configuration register of a timer that is already enabled may result in undefined behavior. For example, merely changing a timer from one-shot to periodic may not produce what is intended. Timers should always be disabled prior to changing any other properties.

12.5.2 **Synthetic Timer Count Register**

63:0
Count

Bits	Description	Attributes
63:0	Count—expiration time for one-shot timers, duration for periodic timers	Read/write

The value programmed into the Count register is a time value measured in 100 nanosecond units. Writing the value zero to the Count register will stop the counter, thereby disabling the timer, independent of the setting of *AutoEnable* in the configuration register.

Note that the Count register is permitted to wrap. Wrapping will have no effect on the behavior of the timer, regardless of any timer property.

For one-shot timers, it represents the absolute timer expiration time. The timer expires when the reference counter for the partition is equal to or greater than the specified count value.

For periodic timers, the count represents the period of the timer. The first period begins when the synthetic timer is enabled.

12.6 Synthetic Time-Unhalted Timer MSRs

Synthetic Time-Unhalted Timer MSRs are available if a partition has the AccessSyntheticTimerRegs privilege and EDX bit 23 in the Hypervisor Feature Identification CPUID leaf 0x40000003 is set. Guest software may program the synthetic time-unhalted timer to generate a periodic interrupt after executing for a specified amount of time in 100ns units. When the interrupt fires, the SyntheticTimeUnhaltedTimerExpired field in the VP Assist Page will be set to TRUE. Guest software may reset this field to FALSE. Unlike architectural performance counters, the synthetic timer is never reset by the hypervisor and runs continuously between interrupts.

Unlike regular synthetic timers that accumulate time when the guest has halted (ie: gone idle), the Synthetic Time-Unhalted Timer accumulates time only while the guest is not halted.

MSR Address	Register Name	Function
0x40000114	HV_X64_MSR_STIME_UNHALTED_TIMER_CONFIG	Synthetic Time-Unhalted Timer Configuration MSR
0x40000115	HV_X64_MSR_STIME_UNHALTED_TIMER_COUNT	Synthetic Time-Unhalted Timer Count MSR

12.6.1 **Synthetic Time-Unhalted Timer Configuation MSR**

63:56	55	54:0
Vector	Enabled	RsvdZ

Bits	Description	Attributes
63:56	Vector	Read/write
55	Enabled	Read/write
54:0	RsvdZ (value should be set to zero)	Read/write

12.6.2 **Synthetic Time-Unhalted Timer Count MSR**

63:0
Count

Bits	Description	Attributes
63:0	Periodic rate of interrupts in 100 ns units	Read/write

12.7 Partition Reference Time Enlightenment

The hypervisor provides a partition-wide virtual reference TSC page which is overlaid on the partition's GPA space. A partition's reference time stamp counter page is accessed through the Reference TSC MSR. A partition which possesses the AccessPartitionReferenceTsc privilege may access the reference TSC MSR.

12.7.1 **Reference Time Stamp Counter (TSC) Page MSR**

A guest wishing to access its reference TSC page must use the following model-specific register (MSR).

MSR Address	Register Name	Function
0x40000021	HV_X64_MSR_REFERENCE_TSC	Time reference count (partition-wide)

The format of the Reference TSC MSR is as follows:

63:12	11:1	0
GPA Page Number	RsvdP	Enable

Bits	Description	Attributes
63:12	GPA Page Number	Read/write
11:1	RsvdP (value should be preserved)	Read/write
0	Enable—set if reference TSC is enabled	Read/write

At the guest partition creation time, the value of the reference TSC MSR is 0x0000000000000000. Thus, the reference TSC page is disabled by default. The guest must enable the reference TSC page by setting bit 0. If the specified base address is beyond the end of the partition's GPA space, the reference TSC page will not be accessible to the guest. When modifying the register, guests should preserve the value of the reserved bits (1 through 11) for future compatibility.

Restrictions

The caller must possess the AccessPartitionReferenceTsc privilege.

12.7.2 **Format of the Reference TSC Page**

The reference TSC page is defined using the following structure:

```
typedef struct _HV_REFERENCE_TSC_PAGE
{
    volatile UINT32     TscSequence;
    UINT32              Reserved1;
    volatile UINT64     TscScale;
    volatile INT64      TscOffset;
    UINT64              Reserved2[509];
} HV_REFERENCE_TSC_PAGE, *PHV_REFERENCE_TSC_PAGE;
```

12.7.3 **Partition Reference TSC Mechanism**

The partition reference time is computed by the following formula:

ReferenceTime = ((VirtualTsc * TscScale) >> 64) + TscOffset

The multiplication is a 64 bit multiplication, which results in a 128 bit number which is then shifted 64 times to the right to obtain the high 64 bits.

12.7.3.1 **TscScale**

The TscScale value is used to adjust the Virtual TSC value across migration events to mitigate TSC frequency changes from one platform to another.

12.7.3.2 **TscSequence**

The TscSequence value is used to synchronize access to the enlightened reference time if the scale and/or the offset fields are changed during save/restore or live migration. This field serves as a sequence number which is incremented whenever the scale and/or the offset fields are modified. A special value of 0x0 is used to indicate that this facility is no longer a reliable source of reference time and the VM must fall back to a different source.

12.7.3.3 **Reference TSC during Save/Restore and Migration**

To address migration scenarios to physical platforms which do not support iTSC, the TscSequence field is used. In the event a guest partition is migrated from an iTSC capable host to a non-iTSC capable host, the hypervisor sets TscSequence to the special value of 0x0, which directs the guest operating system to fall back to a different clock source. The recommended code for computing the partition reference time using this enlightenment is shown below:

```
do {
    StartSequence = ReferenceTscPage->TscSequence;
    if (StartSequence == 0) {
      // 0 means that the Reference TSC enlightenment is not available at
      // the moment, and the Reference Time can only be obtained from
      // reading the Reference Counter MSR.
        ReferenceTime = rdmsr(HV_X64_MSR_TIME_REF_COUNT);
        return ReferenceTime;
    }

    Tsc = rdtsc();
      // Assigning Scale and Offset should neither happen before
      // setting StartSequence, nor after setting EndSequence.
    Scale = ReferenceTscPage->TscScale;
    Offset = ReferenceTscPage->TscOffset;

   EndSequence = ReferenceTscPage->TscSequence;

} while (EndSequence != StartSequence);

// The result of the multiplication is treated as a 128-bit value.
ReferenceTime = ((Tsc * Scale) >> 64) + Offset;
return ReferenceTime;
```

13 Message Formats

13.1 Overview

The hypervisor supports a simple message-based inter-partition communication mechanism. Messages can be sent by the hypervisor to a partition or can be sent from one partition to another. This section describes all of the messages sent by the hypervisor.

Each message has a message type, a source partition, and a message payload. For a complete list of message types, see chapter 11. The format of the message payload depends on the message type.

The messages sent by the hypervisor fall into the following categories:

- Memory access messages (unmapped GPA, GPA access violations, and so on.)
- Processor intercepts
- Error messages
- Timer notifications
- Event log events

13.2 Message Data Types

Intercept messages are delivered by the SynIC.

13.2.1 Message Header

Each message begins with a common message header. The significant fields are the *MessageType*, *PayloadSize* and the *OriginationId* (the source of the message). It is important to note that the payload size reflects only the size of the data and does not include the message header. The message header is described in section 11.10.4.

13.2.2 Intercept Message Header

All x64 memory access messages and processor intercept messages contain a common payload header. This header contains information about the state of the virtual processor at the time of the intercept, making it easier for the recipient of the message to complete the intercepted instruction in software.

```
typedef struct
{
   HV_VP_INDEX    VpIndex;
   UINT8    InstructionLength;
   HV_INTERCEPT_ACCESS_TYPE_MASK    InterceptAccessType;
   HV_X64_VP_EXECUTION_STATE  ExecutionState;
   HV_X64_SEGMENT_REGISTER    CsSegment;
   UINT64   Rip;
   UINT64   Rflags;
} HV_X64_INTERCEPT_MESSAGE_HEADER;
```

VpIndex indicates the index of the virtual processor that generated the intercept.

InstructionLength indicates the byte length of the instruction that generated the intercept. If the instruction length is unknown, a length of zero is reported, and the recipient of the message must fetch and decode the instruction to determine its length. The hypervisor guarantees that it will fill in the correct instruction length for CPUID, I/O port, and MSR intercepts.

InterceptAccessType indicates the access type (read, write, or execute) of the event that triggered the intercept.

ExecutionState provides miscellaneous information about the virtual processor's state at the time the intercept was triggered.

CsSegment provides information about the code segment at the time the intercept was triggered.

Rip provides the instruction pointer at the time the intercept was triggered.

Rflags provides the flags register at the time the intercept was triggered.

13.2.3 **VP Execution State**

The execution state is a collection of flags that specify miscellaneous states of the virtual processor.

```
typedef struct
{
    UINT16 Cpl:2;
    UINT16 Cr0Pe:1;
    UINT16 Cr0Am:1;
    UINT16 EferLma:1;
    UINT16 DebugActive:1;
    UINT16 InterruptionPending:1;
    UINT16 Reserved:4;
    UINT16 Reserved:5;
} HV_X64_VP_EXECUTION_STATE;
```

Cpl indicates the current privilege level at the time of the intercept. Real mode has an implied CPL of 0, and v86 has an implied CPL of 3. In other modes, the CPL is defined by the low-order two bits of the code segment (CS).

Cr0Pe indicates whether the processor is executing within protected mode.

Cr0Am indicates whether alignment must be checked for non-privileged accesses.

EferLma indicates whether the processor is executing within long mode (64-bit mode).

DebugActive indicates that one or more debug registers are marked as active, so the recipient of the message may need to perform additional work to correctly emulate the behavior of the debug breakpoint facilities.

InterruptionPending indicates that the intercept was generated while delivering an interruption. The interruption is held pending and, unless removed, will be re-delivered when the virtual processor is resumed. For a description of the *Pending Interruption* register, see section 7.9.3.

13.2.4 **I/O Port Access Information**

On x64 platforms, I/O port access messages include a collection of flags that provide information about the memory access.

```
typedef struct
{
   UINT8    AccessSize:3;
   UINT8    StringOp:1;
   UINT8    RepPrefix:1;
   UINT8    Reserved:3;
} HV_X64_IO_PORT_ACCESS_INFO;
```

AccessSize indicates the size of the access. The following encodings are used: 001b = 8 bits; 010b = 16 bits; 100b = 32 bits. All other combinations are reserved.

StringOp indicates that the instruction is a string form (INS or OUTS).

RepPrefix indicates that the instruction has a "rep" prefix. This flag is used only for string operations.

13.2.5 **Exception Information**

On x64 platforms, exception intercept messages include a collection of flags that provide information about the exception.

```
typedef struct
{
   UINT8    ErrorCodeValid:1;
   UINT8    Reserved:7;
} HV_X64_EXCEPTION_INFO;
```

ErrorCodeValid indicates that the error code field in the exception message is valid.

13.2.6 **Memory Access Flags**

Memory intercept messages include a collection of flags that provide information about the intercept.

```
typedef struct
{
    UINT8                              GvaValid:1;
    UINT8                              Reserved:7;
} HV_X64_MEMORY_ACCESS_INFO;
```

GvaValid indicates that the *Gva* field of the memory access message contains a valid guest virtual address.

13.3 Timer Messages

13.3.1 **Timer Expiration Message**

Timer expiration messages are sent when a timer event fires.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	Rsvd (8 bytes)		

Timer Expiration Payload	16	TimerIndex (4 bytes)	Rsvd (4 bytes)
	24	ExpirationTime (8 bytes)	
	32	DeliveryTime (8 bytes)	

TimerIndex is the index of the synthetic timer (0 through 3) that generated the message. This allows a client to configure multiple timers to use the same interrupt vector and differentiate between their messages.

ExpirationTime is the expected expiration time of the timer measured in 100-nanosecond units by using the time base of the partition's reference time counter. Note that the expiration message may arrive after the expiration time.

DeliveryTime is the time when the message is placed into the respective message slot of the SIM page. The time is measured in 100-nanosecond units based on the partition's reference time counter.

14 Scheduler

14.1 Scheduling Concepts

The hypervisor schedules virtual processors to run on logical processors.

The hypervisor scheduler makes scheduling decisions based on policy settings set by the root. For further information on Hyper-V scheduling concepts, please reference https://aka.ms/Hyper-VSchedulerDocsTLFS.

14.2 Scheduling Policy Settings

Various scheduler policy settings can be set by the root partition administrator. These include:

- CPU Reserve
- CPU Cap
- CPU Weight

Each of these is described in more detail in the following sections.

14.2.1 CPU Reserve

A CPU reserve can be supplied for each partition. The hypervisor guarantees that this fraction of CPU time is available to each virtual processor within the partition as needed. It does not necessarily mean that the virtual processors *will* consume the entire reserve. If they are idle or waiting on hypervisor work, other virtual processors may consume the available processor cycles.

The CPU reserve is specified as a fraction of a logical processor's capacity. A reserve value of 0.5 indicates that 50% of a logical processor is reserved for each virtual processor in the partition. Valid reserve values range from 0 (no reserve) to 1 (in which case each virtual processor is guaranteed to get 100% of a logical processor if required). The reserve value may not be greater than the CPU Cap (see section 14.2.2).

The reserve value is expressed as an integer ranging from 0x00000000 to 0x00010000. For example, the value 0x0000C000 indicates 0.75, or 75% of a logical processor. By default, a partition's reserve is set to 0x00000000, indicating that there is no reserve.

The total reserves for all virtual processors cannot exceed the number of logical processors in the system. If the number of virtual processors in a partition is greater than the number of logical processors, then reserves are not guaranteed, and the partition's reserve may be reset to zero.

14.2.2 CPU Cap

A CPU cap can be specified for each partition. The hypervisor guarantees that the fraction of CPU time consumed by each virtual processor within the partition will not exceed this cap.

The CPU cap is specified as a fraction of a logical processor's capacity. A cap value of 0.5 indicates that each virtual processor will be restricted to using 50% of a logical processor. Valid cap values range from 0 to 1 (in which case the cap has no effect).

The CPU cap value may not be less than the CPU reserve (see section 14.2.1).

The cap value is expressed as an integer ranging from 0x00000000 to 0x00010000. For example, the value 0x0000C000 indicates 0.75, or 75% of a logical processor.

By default, a partition's cap is set to 0x00010000, indicating that there is no cap.

14.2.3 **CPU Weight**

The CPU weight is a relative weight assigned to each of the virtual processors of the partition. Unless otherwise constrained by reserves and caps, the scheduler will attempt to weight the run time of the virtual processors scheduled on a given logical processor according to their relative weights. Let's consider the case where three partitions, each with one virtual processor are being scheduled on a single logical processor, their weights are 100, 200 and 700, no reserves or caps are in effect, and all three of the virtual processors have work to perform (that is, they are not idle). In this case, the fraction of physical CPU capacity provided to the three virtual processors would be approximately 10%, 20%, and 70%.

The CPU weight value is expressed as a decimal value from 1 to 10,000 where 100 (the geometric mean) is the typical value.

By default, a partition's weight is set to 100.

14.3 Other Scheduling Considerations

14.3.1 **Guest Spinlocks**

A typical multiprocessor-capable operating system uses locks for enforcing atomicity of certain operations. When running such an operating system inside a virtual machine controlled by the hypervisor these critical sections protected by locks can be extended by intercepts generated by the critical section code. The critical section code may also be preempted by the hypervisor scheduler. Although the hypervisor attempts to prevent such preemptions, they can occur. Consequently, other lock contenders could end up spinning until the lock holder is re-scheduled again and therefore significantly extend the spinlock acquisition time. The hypervisor indicates to the guest OS the number of times a spinlock acquisition should be attempted before indicating an excessive spin situation to the hypervisor. This count is returned in CPUID leaf 0x40000004.

The HvNotifyLongSpinWait hypercall provides an interface for enlightened guests to improve the statistical fairness property of a lock for multiprocessor virtual machines. The guest should make this notification on every multiple of the recommended count returned by CPUID leaf 0x40000004. Through this hypercall, a guest notifies the hypervisor of a long spinlock acquisition. This allows the hypervisor to make better scheduling decisions.

14.4 Scheduler Data Types

The following data types support the scheduler interfaces.

```
typedef struct HV_CALL_ATTRIBUTES _HV_INPUT_NOTIFY_LONG_SPINWAIT
{
      UINT32 InitialLongSpinWait;
      UINT32 RsvdZ;
} HV_INPUT_NOTIFY_LONG_SPINWAIT,
```

14.5 Scheduler Interfaces

14.5.1 HvNotifyLongSpinWait

The HvNotifyLongSpinWait hypercall is used by a guest OS to notify the hypervisor that the calling virtual processor is attempting to acquire a resource that is potentially held by another virtual processor within the same partition. This scheduling hint improves the scalability of partitions with more than one virtual processor.

Wrapper Interface

```
HV_STATUS
HvNotifyLongSpinWait(
      _in HV_INPUT_NOTIFY_LONG_SPINWAIT SpinwaitInfo
);
```

Native Interface

HvNotifyLongSpinWait [fast]		
	Call Code = 0x0008	
➡ Input Parameters		
0	SpinwaitInfo (4 bytes)	Padding (4 bytes)

Description

The HvNotifyLongSpinWait hypercall allows a partition to inform the Hypervisor of a long spinlock acquire failure. The hypervisor can use this information to make better scheduling decisions for the notifying virtual processor and its partition.

Input Parameters

SpinwaitInfo – Specifies the accumulated count the guest was spinning.

Output Parameters

None.

Restrictions

None.

Return Values

There is no error status for this hypercall, only HV_STATUS_SUCCESS will be returned as this is an advisory hypercall.

15 Virtual Secure Mode

15.1 Overview

Virtual Secure Mode (VSM) is a set of hypervisor capabilities and enlightenments offered to host and guest partitions which enables the creation and management of new security boundaries within operating system software. VSM is the hypervisor facility on which Windows security features including Device Guard, Credential Guard, virtual TPMs and shielded VMs are based. These security features were introduced in Windows 10 and Windows Server 2016.

VSM enables operating system software in the root and guest partitions to create isolated regions of memory for storage and processing of system security assets. Access to these isolated regions is controlled and granted solely through the hypervisor, which is a highly privileged, highly trusted part of the system's Trusted Compute Base (TCB). Because the hypervisor runs at a higher privilege level than operating system software and has exclusive control of key system hardware resources such as memory access permission controls in the CPU MMU and IOMMU early in system initialization, the hypervisor can protect these isolated regions from unauthorized access, even from operating system software (e.g., OS kernel and device drivers) with supervisor mode access (i.e. CPL0, or "Ring 0").

With this architecture, even if normal system level software running in supervisor mode (e.g. kernel, drivers, etc.) is compromised by malicious software, the assets in isolated regions protected by the hypervisor can remain secured.

15.2 Virtual Trust Levels (VTL)

VSM achieves and maintains isolation through Virtual Trust Levels (VTLs). VTLs are enabled and managed on both a per-partition and per-virtual processor basis.

Virtual Trust Levels are hierarchical, with higher levels being more privileged than lower levels. VTL0 is the least privileged level, with VTL1 being more privileged than VTL0, VTL2 being more privileged than VTL1, etc.

Architecturally, up to 16 levels of VTLs are supported; however a hypervisor may choose to implement fewer than 16 VTL's. Currently, only two VTLs are implemented.

```
//
// Define a virtual trust level (VTL)
//

typedef UINT8 HV_VTL, *PHV_VTL;
#define HV_NUM_VTLS  2
#define HV_INVALID_VTL ((HV_VTL) -1)
#define HV_VTL_ALL 0xF
```

Each VTL has its own set of memory access protections. These access protections are managed by the hypervisor in a partition's physical address space, and thus cannot be modified by system level software running in the partition.

Since more privileged VTLs can enforce their own memory protections, higher VTLs can effectively protect areas of memory from lower VTLs. In practice, this allows a lower VTL to protect isolated memory regions by securing them with a higher VTL. For example, VTL0 could store a secret in VTL1, at which point only VTL1 could access it. Even if VTL0 is compromised, the secret would be safe.

15.2.1 VTL Protections

There are multiple facets to achieving isolation between VTLs:

- **Memory Access Protections:** Each VTL maintains a set of guest physical memory access protections. Software running at a particular VTL can only access memory in accordance with these protections.
- **Virtual Processor State:** Virtual processors maintain separate per-VTL state. For example, each VTL defines a set of a private VP registers. Software running at a lower VTL cannot access the higher VTL's private virtual processor's register state.
- **Interrupts:** Along with a separate processor state, each VTL also has its own interrupt subsystem (local APIC). This allows higher VTLs to process interrupts without risking interference from a lower VTL.
- **Overlay Pages:** Certain overlay pages are maintained per-VTL such that higher VTLs have reliable access. E.g. there is a separate hypercall overlay page per VTL.

15.3 VSM Detection, Enabling, and Status

15.3.1 VSM Detection

The VSM capability is advertised to partitions via the AccessVsm partition privilege flag. Only partitions with all of the following privileges may utilize VSM: AccessVsm, AccessVpRegisters, and AccessSynicRegs. See Partition Privilege Flags.

15.3.2 VSM Capability Detection

Guests should use the following model-specific register to access a report on VSM capabilities:

MSR address	Register name	Function
0x000D0006	HV_X64_REGISTER_VSM_CAPABILITIES	Report on VSM capabilities

The format of the Register VSM Capabilities MSR is as follows:

63	62:47	46	45:0
Dr6Shared	MbecVtlMask	Deny lower VTL Startup	ReservedZ

Bits	Description	Attributes
63	Dr6Shared	Read
62:47	MbecVtlMask	Read

Bits	Description	Attributes
46	DenyLowerVtlStartup	Read
45:0	ReservedZ	Read

Dr6Shared indicates to the guest whether Dr6 is a shared register between the VTLs.

MvecVtlMask indicates to the guest the VTLs for which Mbec can be enabled.

DenyLowerVtlStartup indicates to the guest whether a Vtl can deny a VP reset by a lower VTL. Please reference Section 15.5 for more information.

15.3.3 **VSM Status Registers**

In addition to a partition privilege flag, two virtual registers can be used to learn additional information about VSM status: `HvRegisterVsmPartitionStatus` and `HvRegisterVsmVpStatus`.

15.3.3.1 **HvRegisterVsmPartitionStatus**

HvRegisterVsmPartitionStatus is a per-partition read-only register that is shared across all VTLs. This register provides information about which VTLs have been enabled for the partition, which VTLs have Mode Based Execution Controls enabled, as well as the maximum VTL allowed.

```
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 EnabledVtlSet    : 16;
        UINT64 MaximumVtl       : 4;
        UINT64 MbecEnabledVtlSet: 16;
        UINT64 ReservedZ        : 28;
    };
} HV_REGISTER_VSM_PARTITION_STATUS;
```

15.3.3.2 **HvRegisterVsmVpStatus**

HvRegisterVsmVpStatus is a read-only register and is shared across all VTLs. It is a per-VP register, meaning each virtual processor maintains its own instance. This register provides information about which VTLs have been enabled, which is active, as well as the MBEC mode active on a VP.

```
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 ActiveVtl            : 4;
        UINT64 ActiveMbecEnabled    : 1;
        UINT64 ReservedZ0           : 11;
        UINT64 EnabledVtlSet        : 16;
        UINT64 ReservedZ1           : 32;
    };
} HV_REGISTER_VSM_VP_STATUS;
```

ActiveVtl is the ID of the VTL context that is currently active on the virtual processor.

ActiveMbecEnabled specifies that MBEC is currently active on the virtual processor.

EnabledVtlSet is a bitmap of the VTL's that are enabled on the virtual processor.

15.3.4 **Partition VTL Initial state**

When a partition starts or resets, it begins running in VTL0. All other VTLs are disabled at partition creation.

15.4 VTL Enablement

To begin using a VTL, a lower VTL must initiate the following:

1) Enable the target VTL for the partition. This makes the VTL generally available for the partition.
2) Enable the target VTL on one or more virtual processors. This makes the VTL available for a VP, and sets its initial context. It is recommended that all VPs have the same enabled VTLs. Having a VTL enabled on some VPs (but not all) can lead to unexpected behavior.
3) Once the VTL is enabled for a partition and VP, it can begin setting access protections once the EnableVtlProtection flag has been set (see 15.5.1.1).

Note that VTLs need not be consecutive.

15.4.1 **Enabling a Target VTL for a Partition**

The HvCallEnablePartitionVtl hypercall is used to enable a VTL for a certain partition. Note that before software can actually execute in a particular VTL, that VTL must be enabled on virtual processors in the partition.

15.4.2 **Enabling a Target VTL for Virtual Processors**

Once a VTL is enabled for a partition, it can be enabled on the partition's virtual processors. The HvCallEnableVpVtl hypercall can be used to enable VTLs for a virtual processor, which sets its initial context.

Virtual processors have one "context" per VTL. If a VTL is switched, the virtual processor context is also switched. See 15.11 for details on what state is switched.

15.5 VTL Configuration

Once a VTL has been enabled, its configuration can be changed by a VP running at an equal or higher VTL.

15.5.1 **Partition Configuration**

Partition-wide attributes can be configured using the HvRegisterVsmPartitionConfig register. There is one instance of this register for each VTL (greater than 0) on every partition.

Every VTL can modify its own instance of HV_REGISTER_VSM_PARTITION_CONFIG, as well as instances for lower VTLs. VTLs may not modify this register for higher VTLs.

```
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 EnableVtlProtection              : 1;
        UINT64 DefaultVtlProtectionMask         : 4;
        UINT64 ZeroMemoryOnReset                : 1;
        UINT64 DenyLowerVtlStartup              : 1;
        UINT64 ReservedZ                        : 2;
        UINT64 InterceptVpStartup               : 1;
        UINT64 ReservedZ                        : 54;    };

} HV_REGISTER_VSM_PARTITION_CONFIG;
```

The fields of this register are described below.

15.5.1.1 **Enable VTL Protections**

Once a VTL has been enabled, the EnableVtlProtection flag must be set before it can begin applying memory protections.

This flag is write-once, meaning that once it has been set, it cannot be modified.

15.5.1.2 **Default Protection Mask**

By default, the system applies RWX protections to all currently mapped pages, and any future "hot-added" pages. Hot-added pages refer to any memory that is added to a partition during a resize operation. See section 15.9 for a description of memory access protections.

A higher VTL can set a different default memory protection policy by specifying DefaultVtlProtectionMask in HV_REGISTER_VSM_PARTITION_CONFIG. This mask must be set at the time the VTL is enabled. It cannot be changed once it is set, and is only cleared by a partition reset.

DefaultVtlProtectionMask
Bit 0: Read
Bit 1: Write
Bit 2: User Mode Execute (UMX)
Bit 3: Kernel Mode Execute (KMX)

15.5.1.3 **Zero Memory on Reset**

ZeroMemOnReset is a bit that controls if memory is zeroed before a partition is reset. This configuration is on by default. If the bit is set, the partition's memory is zeroed upon reset so that a higher VTL's memory cannot be compromised by a lower VTL.

If this bit is cleared, the partition's memory is not zeroed on reset.

To unlock the TLB, the higher VTL can clear this bit. Also, once a VP returns to a lower VTL, it releases all TLB locks which it holds at the time.

15.5.1.4 **DenyLowerVtlStartup**

The DenyLowerVtlStartup flag controls if a virtual processor may be started or reset by lower VTLs. This includes architectural ways of resetting a virtual processor (e.g. SIPI on X64) as well as the HvCallStartVirtualProcessor hypercall (reference section 7.10.3).

15.5.1.5 **InterceptVpStartup**

If InterceptVpStartup flag is set, starting or resetting a virtual processor generates an intercept to the higher VTL.

15.5.2 **Configuring Lower VTLs**

The following register can be used by higher VTLs to configure the behavior of lower VTLs:

```
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 MbecEnabled    : 1;
        UINT64 TlbLocked      : 1;
        UINT64 ReservedZ      : 62;
    };

} HV_REGISTER_VSM_VP_SECURE_VTL_CONFIG;
```

Each VTL (higher than 0) has an instance of this register for every VTL lower than itself. For example, VTL2 would have two instances of this register – one for VTL1, and a second for VTL0.

15.5.2.1 **MbecEnabled**

This field configures whether MBEC is enabled for the lower VTL (see 15.10).

15.5.2.2 **TlbLocked**

This field locks the lower VTL's TLB. This capability can be used to prevent lower VTLs from causing TLB invalidations which might interfere with a higher VTL. When this bit is set, all address space flush requests from the lower VTL are blocked until the lock is lifted.

15.6 VTL Entry

A VTL is "entered" when a VP switches from a lower VTL to a higher one. This can happen for the following reasons:

1. **VTL call:** this is when software explicitly wishes to invoke code in a higher VTL.

2. **Secure interrupt:** if an interrupt is received for a higher VTL, the VP will enter the higher VTL. See 15.12.
3. **Secure intercept**: certain actions will trigger a secure interrupt (accessing certain MSRs for example). See 15.13.

Once a VTL is entered, it must voluntarily exit. A higher VTL cannot be preempted by a lower VTL.

15.6.1 **VTL Call**

A "VTL call" is when a lower VTL initiates an entry into a higher VTL (for example, to protect a region of memory with the higher VTL).

VTL calls preserve the state of shared registers across VTL switches. Private registers are preserved on a per-VTL level. (See 15.11.1 and 15.11.2 for which state is shared/private). The exception to these restrictions are the registers required by the VTL call sequence. The following registers are required for a VTL call:

x64 Register	x86 Register	Value	Description
RCX	EDX:EAX	RsvdZ	Specifies a VTL call control input to the hypervisor
RAX	ECX		Reserved

15.6.1.1 **VTL Call Restrictions**

VTL calls can only be initiated from the most privileged processor mode. For example, on x64 systems a VTL call can only come from CPL0. A VTL call initiated from a processor mode which is anything but the most privileged on the system results in the hypervisor injecting a #UD exception into the virtual processor.

A VTL call can only switch into the next highest VTL. In other words, if there are multiple VTLs enabled, a call cannot "skip" a VTL.

The following actions result in a #UD exception:

1. A VTL call initiated from a processor mode which is anything but the most privileged on the system (architecture specific).
2. A VTL call from real mode (x86/x64)
3. A VTL call on a virtual processor where the target VTL is disabled (or has not been already enabled).
4. A VTL call with an invalid control input value

15.6.1.2 **Identifying VTL Entry Reason**

In order to react appropriately to an entry, a higher VTL might need to know the reason it was entered. To discern between entry reasons, the following field is included in the VP assist page (see 7.8.7):

```
typedef enum
{
    // This reason is reserved and is not used.
    HvVtlEntryReserved             = 0,

    // Indicates entry due to a VTL call from a lower VTL.
    HvVtlEntryVtlCall              = 1,

    // Indicates entry due to an interrupt targeted to the VTL.
    HvVtlEntryInterrupt            = 2

} HV_VTL_ENTRY_REASON;
```

15.7 VTL Exit

A switch to a lower VTL is known as a "return". Once a VTL has finished processing, it can initiate a VTL return in order to switch to a lower VTL. The only way a VTL return can occur is if a higher VTL voluntarily initiates one. A lower VTL can never preempt a higher one.

15.7.1 **VTL Return**

A "VTL return" is when a higher VTL initiates a switch into a lower VTL. Similar to a VTL call, private processor state is switched out, and shared state remains in place (See 15.11.1 and 15.11.2 for which state is shared/private). If the lower VTL has explicitly called into the higher VTL, the hypervisor increments the higher VTL's instruction pointer before the return is complete so that it may continue after a VTL call.

A VTL Return code sequence requires the use of the following registers:

x64 Register	x86 Register	Register Value	Description
RCX	EDX:EAX	Bits 63:1 - RsvdZ Bit 0 - Fast return (See 15.7.1.1)	Specifies a VTL return control input to the hypervisor
RAX	ECX		Reserved

15.7.1.1 **Fast return**

As a part of processing a return, the hypervisor can restore the lower VTL's register state from the VTL control page. For example, after processing a secure interrupt, a higher VTL may wish to return without disrupting the lower VTL's state. Therefore, the hypervisor provides a mechanism to simply restore the lower VTL's registers to their pre-call value stored in the control page.

If this behavior is not necessary, a higher VTL can use a "fast return". A fast return is when the hypervisor does not restore register state from the control page. This should be utilized whenever possible to avoid unnecessary processing.

This field can be set with bit 0 of the VTL return input. If it is set to 0, the registers are restored from the VP assist page. If this bit is set to 1, the registers are not restored (a fast return).

15.7.2 **Restrictions**

The following actions will generate a #UD exception:

1. Attempting a VTL return when the lowest VTL is currently active
2. Attempting a VTL return with an invalid control input value
3. Attempting a VTL return from a processor mode which is anything but the most privileged on the system (architecture specific)

15.8 Hypercall Page Assists

The hypervisor provides mechanisms to assist with VTL calls and returns via the hypercall page (see 3.13). This page abstracts the specific code sequence required to switch VTLs.

The code sequences to execute VTL calls and returns may be accessed by executing specific instructions in the hypercall page. The call/return chunks are located at an offset in the hypercall page determined by the HvRegisterVsmCodePageOffset virtual register. This is a read-only and partition-wide register, with a separate instance per-VTL.

A VTL can execute a VTL call/return using the CALL instruction. A CALL to the correct location in the hypercall page will initiate a VTL call/return.

```
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 VtlCallOffset   : 12;
        UINT64 VtlReturnOffset : 12;
        UINT64 ReservedZ       : 40;
    };
} HV_REGISTER_VSM_CODE_PAGE_OFFSETS;
```

To summarize, the steps for calling a code sequence using the hypercall page are as follows:

1. Map the hypercall page into a VTL's GPA space (see 3.13).
2. Determine the correct offset for the code sequence (VTL call or return).
3. Execute the code sequence using CALL.

15.8.1 VTL Control via the VP Assist Page

The hypervisor uses part of the VP assist page to facilitate communication with code running in a VTL higher than VTL0 (see 7.8.7). Each VTL has its own control structure (except VTL0).

15.8.2 Definition

The following information is communicated using the control page:

1. The VTL entry reason.
2. A flag indicating that VINA is being asserted.
3. The values for registers to load upon a VTL return.

```
typedef struct
{
    // The hypervisor updates the entry reason with an indication as to why
    // the VTL was entered on the virtual processor.
    HV_VTL_ENTRY_REASON     EntryReason;

    // This flag determines whether the VINA interrupt line is asserted.
    union
    {
        UINT8               AsUINT8;
        struct
        {
            UINT8           VinaAsserted  :1;
            UINT8           VinaReservedZ :7;
        };
    } VinaStatus;

    UINT8                   ReservedZ00;
    UINT16                  ReservedZ01;

    // A guest updates the VtlReturn* fields to provide the register values
    // to restore on VTL return.  The specific register values that are
    // restored will vary based on whether the VTL is 32-bit or 64-bit.

    union
    {
        struct
        {
            UINT64          VtlReturnX64Rax;
            UINT64          VtlReturnX64Rcx;
        };

        struct
        {
            UINT32          VtlReturnX86Eax;
            UINT32          VtlReturnX86Ecx;
            UINT32          VtlReturnX86Edx;
            UINT32          ReservedZ1;
        };
    };

} HV_VP_VTL_CONTROL;
```

15.9 Memory Access Protections

One necessary protection provided by VSM is the ability to isolate memory accesses.

15.9.1 **Memory Protection Hierarchy**

Memory access permissions can be set by a number of sources for a particular VTL. Each VTL's permissions can potentially be restricted by a number of other VTLs, as well as by the host partition. The order in which protections are applied is the following:

1. Memory protections set by the host
2. Memory protections set by higher VTLs

In other words, VTL protections supersede host protections. Higher-level VTLs supersede lower-level VTLs. Note that a VTL may not set memory access permissions for itself.

A conformant interface is expected to not overlay any non-RAM type over RAM.

15.9.2 **Memory Access Violations**

If a VP running at a lower VTL attempts to violate a memory protection set by a higher VTL, an intercept is generated. This intercept is received by the higher VTL which set the protection. This allows higher VTLs to deal with the violation on a case-by-case basis. For example, the higher VTL may choose to return a fault, or emulate the access (see 15.13)

15.9.3 **Default memory protection types**

Higher VTLs have a high degree of control over the type of memory access permissible by lower VTLs. There are three basic types of protections that can be specified by a higher VTL for a particular GPA page: Read, Write, and eXecute. These are defined in the following table:

Name	**Description**
Read	Controls whether read access is allowed to a memory page
Write	Controls whether write access allowed to a memory page
Execute	Controls whether instruction fetches are allowed for a memory page

These three combine for the following types of memory protection:

1. No access
2. Read-only, no execute
3. Read-only, execute
4. Read/write, no execute
5. Read/write, execute

These three types will continue to be the only memory protections supported for use by the host OS when restricting the guest. However, a VTL mask will have additional memory protections available to restrict lower VTL. This capability is known as “mode based execution control (MBEC)”.

15.10 Mode Based Execution Control (MBEC)

When a VTL places a memory restriction on a lower VTL, it may wish to make a distinction between user and kernel mode when granting an “execute” privilege. For example, if code integrity checks were to take place in a higher VTL, the ability to distinguish between user-mode and kernel-mode would mean that a VTL could enforce code integrity for only kernel-mode applications.

Apart from the traditional three memory protections (read, write, execute), MBEC introduces a distinction between user-mode and kernel-mode for execute protections. Thus, if MBEC is enabled, a VTL has the opportunity to set four types of memory protections:

Name	Description
Read	Controls whether read access is allowed to a memory page
Write	Controls whether write access allowed to a memory page
User Mode Execute (UMX)	Controls whether instruction fetches generated in user-mode are allowed for a memory page NOTE: If MBEC is disabled, this setting is ignored.
Kernel Mode Execute (KMX)	Controls whether instruction fetches generated in kernel-mode are allowed for a memory page NOTE: If MBEC is disabled, this setting controls both user-mode and kernel-mode execute accesses.

Memory marked with the “User-Mode Execute” protections would only be executable when the virtual processor is running in user-mode. Likewise, “Kernel-Mode Execute” memory would only be executable when the virtual processor is running in kernel-mode.

KMX and UMX can be independently set such that execute permissions are enforced differently between user and kernel mode. All combinations of UMX and KMX are supported, except for KMX=1, UMX=0. The behavior of this combination is undefined.

MBEC is disabled by default for all VTLs and virtual processors. When MBEC is disabled, the kernel-mode execute bit determines memory access restriction. Thus, if MBEC is disabled, KMX=1 code is executable in both kernel and user-mode.

15.10.1 **Descriptor Tables**

Any user-mode code that accesses descriptor tables must be in GPA pages marked as KMX=UMX=1. User-mode software accessing descriptor tables from a GPA page marked KMX=0 is unsupported and results in a general protection fault.

15.10.2 **MBEC configuration**

To make use of Mode-based execution control, it must be enabled at two levels:

1. When the VTL is enabled for a partition, MBEC must be enabled using `HvEnablePartitionVtl` (see 15.15.3).
2. MBEC must be configured on a per-VP and per-VTL basis, using `HvRegisterVsmVpSecureVtlConfig` (see 15.5.2).

15.10.3 **MBEC Interaction with Supervisor Mode Execution Prevention (SMEP)**

Supervisor-Mode Execution Prevention (SMEP) is a processor feature supported on some platforms. SMEP can impact the operation of MBEC due to its restriction of supervisor access to memory pages. The hypervisor adheres to the following policies related to SMEP:

1. If SMEP is not available to the guest OS (whether it be because of hardware capabilities or processor compatibility mode), MBEC operates unaffected.

2. If SMEP is available, and is **enabled**, MBEC operates unaffected.
3. If SMEP is **available**, and is **disabled**, all execute restrictions are governed by the KMX control. Thus, only code marked KMX=1 will be allowed to execute.

15.11 Virtual Processor State Isolation

Virtual processors maintain separate states for each active VTL. However, some of this state is private to a particular VTL, and the remaining state is shared among all VTLs.

State which is preserved per VTL (a.k.a. private state) is saved by the hypervisor across VTL transitions. If a VTL switch is initiated, the hypervisor saves the current private state for the active VTL, and then switches to the private state of the target VTL. Shared state remains active regardless of VTL switches.

15.11.1 **Private State**

In general, each VTL has its own control registers, RIP register, RSP register, and MSRs. Below is a list of specific registers and MSRs which are private to each VTL.

15.11.1.1 **Private MSRs**

SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP, STAR, LSTAR, CSTAR, SFMASK, EFER, PAT, KERNEL_GSBASE, FS.BASE, GS.BASE, TSC_AUX
HV_X64_MSR_HYPERCALL
HV_X64_MSR_GUEST_OS_ID
HV_X64_MSR_REFERENCE_TSC
HV_X64_MSR_APIC_FREQUENCY
HV_X64_MSR_EOI
HV_X64_MSR_ICR
HV_X64_MSR_TPR
HV_X64_MSR_APIC_ASSIST_PAGE
HV_X64_MSR_NPIEP_CONFIG
HV_X64_MSR_SIRBP
HV_X64_MSR_SCONTROL
HV_X64_MSR_SVERSION
HV_X64_MSR_SIEFP
HV_X64_MSR_SIMP
HV_X64_MSR_EOM
HV_X64_MSR_SINT0 – HV_X64_MSR_SINT15
HV_X64_MSR_STIMER0_CONFIG – HV_X64_MSR_STIMER3_CONFIG
HV_X64_MSR_STIMER0_COUNT – HV_X64_MSR_STIMER3_COUNT
Local APIC registers (including CR8/TPR)

15.11.1.2 Private registers

RIP, RSP
RFLAGS
CR0, CR3, CR4
DR7
IDTR, GDTR
CS, DS, ES, FS, GS, SS, TR, LDTR
TSC
DR6 (*dependent on processor type. Read `HvRegisterVsmCapabilities` virtual register to determine shared/private status)

15.11.2 Shared State

VTLs share state in order to cut down on the overhead of switching contexts. Sharing state also allows some necessary communication between VTLs. Most general purpose and floating point registers are shared, as are most architectural MSRs. Below is the list of specific MSRs and registers that are shared among all VTLs:

15.11.2.1 Shared MSRs

HV_X64_MSR_TSC_FREQUENCY
HV_X64_MSR_VP_INDEX
HV_X64_MSR_VP_RUNTIME
HV_X64_MSR_RESET
HV_X64_MSR_TIME_REF_COUNT
HV_X64_MSR_GUEST_IDLE
HV_X64_MSR_DEBUG_DEVICE_OPTIONS
HV_X64_MSR_BELOW_1MB_PAGE
HV_X64_MSR_STATS_PARTITION_RETAIL_PAGE
HV_X64_MSR_STATS_VP_RETAIL_PAGE
MTRRs
MCG_CAP
MCG_STATUS

15.11.2.2 Shared registers

Rax, Rbx, Rcx, Rdx, Rsi, Rdi, Rbp
CR2
R8 – R15

DR0 – DR5
X87 floating point state
XMM state
AVX state
XCR0 (XFEM)
DR6 (*dependent on processor type. Read `HvRegisterVsmCapabilities` virtual register to determine shared/private status)

15.11.3 **Real Mode**

Real mode is not supported for any VTL greater than 0. VTLs greater than 0 can run in 32-bit or 64-bit mode. In order to switch between 32-bit and 64-bit in a non-zero VTL, the VTL needs to be disabled and re-initialized.

15.12 VTL Interrupt Management

15.12.1 **Overview**

In order to achieve a high level of isolation between Virtual Trust Levels, Virtual Secure Mode provides a separate interrupt subsystem for each VTL enabled on a virtual processor. This ensures that a VTL is able to both send and receive interrupts without interference from a less secure VTL.

Each VTL has its own interrupt controller, which is only active if the virtual processor is running in that particular VTL. If a virtual processor switches VTL states, the interrupt controller active on the processor is also switched.

15.12.2 **Interrupts targeted at a higher VTL**

An interrupt targeted at a VTL which is higher than the active VTL will cause an immediate VTL switch. The higher VTL can then receive the interrupt. If the higher VTL is unable to receive the interrupt because of its TPR/CR8 value, the interrupt is held as “pending” and the VTL does not switch. If there are multiple VTLs with pending interrupts, the highest VTL takes precedence (without notice to the lower VTL).

15.12.2.1 **RFLAGS.IF**

For the purposes of switching VTLs, RFLAGS.IF does not affect whether a secure interrupt triggers a VTL switch. If RFLAGS.IF is cleared to mask interrupts, interrupts into higher VTLs will still cause a VTL switch to a higher VTL. Only the higher VTL’s TPR/CR8 value is taken into account when deciding whether to immediately interrupt.

This behavior also affects pending interrupts upon a VTL return. If the RFLAGS.IF bit is cleared to mask interrupts in a given VTL, and the VTL returns (to a lower VTL), the hypervisor will reevaluate any pending interrupts. This will cause an immediate call back to the higher VTL.

15.12.3 **Interrupts Targeted at a Lower VTL**

When an interrupt is targeted at a lower VTL, the interrupt is not delivered until the next time the virtual processor transitions into the targeted VTL.

INIT and startup IPIs targeted at a lower VTL are dropped on a virtual processor with a higher VTL enabled. Since INIT/SIPI is blocked, the HvStartVirtualProcessor and HvGetVpIndexFromApicId hypercalls should be used to start processors (see 7.10.3 and 7.10.4, respectively).

15.12.4 **Virtual Interrupt Notification Assist**

Higher VTLs may register to receive a notification if they are blocking immediate delivery of an interrupt to a lower VTL of the same virtual processor. Higher VTLs can enable Virtual Interrupt Notification Assist (VINA) via a virtual register `HvRegisterVsmVina`:

```
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Vector           : 8;
        UINT64 Enabled          : 1;
        UINT64 AutoReset        : 1;
        UINT64 AutoEoi          : 1;
        UINT64 ReservedP        : 53;
    };
} HV_REGISTER_VSM_VINA;
```

Each VTL on each VP has its own VINA instance, as well as its own version of `HvRegisterVsmVina`. The VINA facility will generate an edge triggered interrupt to the currently active higher VTL when an interrupt for the lower VTL is ready for immediate delivery.

In order to prevent a flood of interrupts occurring when this facility is enabled, the VINA facility includes some limited state. When a VINA interrupt is generated, the VINA facility's state is changed to "Asserted." Sending an end-of-interrupt to the SINT associated with the VINA facility will not clear the "Asserted" state. The asserted state can only be cleared in one of two ways:

1) The state can manually be cleared by writing to the `VinaAsserted` field of the VTL control page.
2) The state is automatically cleared on the next entry to the VTL if the "auto-reset on VTL entry" option is enabled in the `HvRegisterVsmVina` register.

This allows code running at a secure VTL to just be notified of the first interrupt that is received for a lower VTL. If a secure VTL wishes to be notified of additional interrupts, it can clear the `VinaAsserted` field of the VP assist page, and it will be notified of the next new interrupt.

15.13 Secure Intercepts

Recall that the root partition has the ability to install an intercept on certain guest actions. With this capability, anytime a specified event takes place in a guest partition, the guest VP is suspended and an intercept is triggered. This allows the root to take action in response to an event (e.g. a memory access).

In much the same way, the hypervisor allows a higher VTL to install intercepts for events that take place in the context of a lower VTL. This gives higher VTLs an elevated level of control over lower-VTL resources. Secure intercepts can be used to protect system-critical resources, and prevent attacks from lower-VTLs.

A normal intercept suspends the guest VP, and sends an intercept message to the root partition. In the case of a secure intercept, the intercept is queued to the higher VTL, and that VTL is made runnable on the VP.

15.13.1 **Secure Intercept Types**

Intercept Type	Intercept Applies To
Memory access	Attempting to access GPA protections established by a higher VTL.
Control register access	Attempting to access a set of control registers specified by a higher VTL.

15.13.2 **Nested Intercepts**

Intercepts can intersect in two cases:

1. Multiple VTLs can install secure intercepts for the same event in a lower VTL.
2. The host and a higher VTL can install an intercept for the same event in a guest.

Thus, a hierarchy is established to decide where nested intercepts are notified. The following list is the order of where intercept are notified:

1. Lower VTL
2. Higher VTL
3. Root partition

A lower VTL always takes precedence, and VTLs always take precedence over the root partition. Once the VTL or root partition is notified of the intercept, no other VTLs or partitions are notified.

15.13.3 **Handling Secure Intercepts**

Once a VTL has been notified of a secure intercept, it must take action such that the lower VTL can continue.

The higher VTL can handle the intercept in a number of ways, including: injecting an exception, emulating the access, or providing a proxy to the access. In any case, if the private state of the lower VTL VP needs to be modified, `HvSetVpRegisters` should be used.

15.13.4 **Secure Register Intercepts**

A higher VTL can intercept on accesses to certain control registers. This is achieved by setting `HvX64RegisterCrInterceptControl` using the `HvSetVpRegisters` hypercall (see 7.10.1).

Setting the control bit in `HvX64RegisterCrInterceptControl` will trigger an intercept for every access of the corresponding control register.

```
HvX64RegisterCrInterceptControl            = 0x000E0000,

typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Cr0Write                     : 1;       // 0x0000000000000001
        UINT64 Cr4Write                     : 1;       // 0x0000000000000002
        UINT64 XCr0Write                    : 1;       // 0x0000000000000004
        UINT64 IA32MiscEnableRead           : 1;       // 0x0000000000000008
        UINT64 IA32MiscEnableWrite          : 1;       // 0x0000000000000010
        UINT64 MsrLstarRead                 : 1;       // 0x0000000000000020
        UINT64 MsrLstarWrite                : 1;       // 0x0000000000000040
        UINT64 MsrStarRead                  : 1;       // 0x0000000000000080
        UINT64 MsrStarWrite                 : 1;       // 0x0000000000000100
        UINT64 MsrCstarRead                 : 1;       // 0x0000000000000200
        UINT64 MsrCstarWrite                : 1;       // 0x0000000000000400
        UINT64 ApicBaseMsrRead              : 1;       // 0x0000000000000800
        UINT64 ApicBaseMsrWrite             : 1;       // 0x0000000000001000
        UINT64 MsrEferRead                  : 1;       // 0x0000000000002000
        UINT64 MsrEferWrite                 : 1;       // 0x0000000000004000
        UINT64 GdtrWrite                    : 1;       // 0x0000000000008000
        UINT64 IdtrWrite                    : 1;       // 0x0000000000010000
        UINT64 LdtrWrite                    : 1;       // 0x0000000000020000
        UINT64 TrWrite                      : 1;       // 0x0000000000040000
        UINT64 MsrSysenterCsWrite           : 1;       // 0x0000000000080000
        UINT64 MsrSysenterEipWrite          : 1;       // 0x0000000000100000
        UINT64 MsrSysenterEspWrite          : 1;       // 0x0000000000200000
        UINT64 MsrSfmaskWrite               : 1;       // 0x0000000000400000
        UINT64 MsrTscAuxWrite               : 1;       // 0x0000000000800000
        UINT64 MsrSgxLaunchControlWrite     : 1;       // 0x0000000001000000
        UINT64 RsvdZ                        : 39;
    };
} HV_REGISTER_CR_INTERCEPT_CONTROL;
```

15.13.4.1 **Mask Registers**

```
HvX64RegisterCrInterceptCr0Mask            = 0x000E0001,
HvX64RegisterCrInterceptCr4Mask            = 0x000E0002,
HvX64RegisterCrInterceptIa32MiscEnableMask = 0x000E0003,
```

To allow for finer control, a subset of control registers also have corresponding mask registers (defined above). Mask registers can be used to install intercepts on a subset of the corresponding control registers. Where a mask register is not defined, any access (as defined by `HvX64RegisterCrInterceptControl`) will trigger an intercept.

15.14 DMA and Devices

Devices effectively have the same privilege level as VTL0. When VSM is enabled, all device-allocated memory is marked as VTL0. Any DMA accesses have the same privileges as VTL0.

15.15 VSM Interfaces

15.15.1 **Type Definitions**

Below are type definitions for a VTL, as well as the input for targeting a VTL with a hypercall.

```
// VTL definition
typedef UINT8 HV_VTL;

// Input for targeting a specific VTL.

typedef union
{
    UINT8 AsUINT8;
    struct
    {
        UINT8 TargetVtl    : 4;
        UINT8 UseTargetVtl : 1;
        UINT8 ReservedZ    : 3;
    };
} HV_INPUT_VTL;
```

15.15.2 **HvModifyVtlProtectionMask**

The HvModifyVtlProtectionMask hypercall modifies the VTL protections applied to an existing set of GPA pages.

Wrapper Interface

```
HV_STATUS
HvModifyVtlProtectionMask (
   _in HV_PARTITION_ID        TargetPartitionId,
   _in HV_MAP_GPA_FLAGS       MapFlags,
   _in HV_INPUT_VTL           TargetVtl,
   _in_ecount(PageCount)      HV_GPA_PAGE_NUMBER GpaPageList
);
```

Native Interface

HvModifyVtlProtectionMask [rep]			
	Call Code = 0x000c		
➡ Input Parameters			
0	TargetPartitionId (8 bytes)		
8	MapFlags(4 bytes)	TargetVTL (1 byte)	RsvdZ (3 bytes)
➡ Input List Element			
0	GpaPageList		

Input Parameters

*TargetPartitionId s*upplies the partition ID of the partition this request is for.

MapFlags specifies the new mapping flags to apply

TargetVtl specifies the VTL to be enabled by this hypercall.

GpaPageList supplies the pages to be protected

Output Parameters

None.

Restrictions

A VTL can only place protections on a lower VTL.

Any attempt to apply VTL protections on non-RAM ranges will fail with HV_STATUS_INVALID_PARAMETER.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller does not possess the CpuManagement privilege.
HV_STATUS_INVALID_PARTITION_ID	A partition with the specified partition Id does not exist.
HV_STATUS_INVALID_PARTITION_STATE	The hypervisor could not perform the operation because the partition is entering or in an invalid state.
HV_STATUS_INVALID_PARAMETER	The hypervisor could not perform the operation because an invalid parameter was specified.
HV_STATUS_OPERATION_DENIED	The operation could not be performed. (The actual cause depends on the operation.)
HV_STATUS_INSUFFICIENT_MEMORY	Insufficient memory exists for the call to succeed.
HV_STATUS_HYPERCALL_INTERCEPT	The requested access to the hypercall generated an intercept.
HV_STATUS_INVALID_VTL_STATE	The supplied virtual trust level is not in the correct state to perform the requested operation.

15.15.3 **HvEnablePartitionVtl**

The HvEnablePartitionVtl hypercall enables a virtual trust level for a specified partition. It should be used in conjunction with HvEnableVpVtl to initiate and use a new VTL.

Wrapper Interface

```
HV_STATUS
HvEnablePartitionVtl (
   _in HV_PARTITION_ID        TargetPartitionId,
   _in HV_VTL                 TargetVtl,
   _in HV_ENABLE_PARTITION_VTL_FLAGS   Flags
);
```

Input Structures

```
typedef union {
    UINT8 AsUINT8;
    struct
    {
        UINT8 EnableMbec:1;
        UINT8 Reserved:7;
    };
} HV_ENABLE_PARTITION_VTL_FLAGS;
```

Native Interface

HvEnablePartitionVtl			
	Call Code = 0x000d		
➡ Input Parameters			
0	TargetPartitionId (8 bytes)		
8	TargetVTL (1 bytes)	Flags (1 byte)	RsvdZ (6 bytes)

Input Parameters

*TargetPartitionId s*upplies the partition ID of the partition this request is for.

Flags specifies a mask to enable VSM related features.

TargetVtl specifies the VTL to be enabled by this hypercall.

Output Parameters

None.

Restrictions

1. A launching VTL can always enable a target VTL if the target VTL is lower than the launching VTL.
2. A launching VTL can enable a higher target VTL if the launching VTL is the highest VTL enabled for the partition that is lower than the target VTL.

These restrictions are only enforced for cases where a partition is launching a VTL for itself. If the host partition is enabling a VTL for a child partition, these restrictions do not apply.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller does not possess the CpuManagement privilege.
HV_STATUS_INVALID_PARTITION_ID	A partition with the specified partition Id does not exist.
HV_STATUS_INVALID_PARAMETER	The hypervisor could not perform the operation because an invalid parameter was specified.
HV_STATUS_INVALID_PARTITION_STATE	The hypervisor could not perform the operation because the partition is entering or in an invalid state.
HV_STATUS_INSUFFICIENT_MEMORY	Insufficient memory exists for the call to succeed.
HV_STATUS_HYPERCALL_INTERCEPT	The requested access to the hypercall generated an intercept.
HV_STATUS_INVALID_VTL_STATE	The supplied virtual trust level is not in the correct state to perform the requested operation.

15.15.4 **HvEnableVpVtl**

HvEnableVpVtl enables a VTL to run on a VP. This hypercall should be used in conjunction with HvEnablePartitionVtl to enable and use a VTL. To enable a VTL on a VP, it must first be enabled for the partition. This call does not change the active VTL.

Wrapper Interface

```
HV_STATUS
HvEnableVpVtl(
   _in HV_PARTITION_ID        TargetPartitionId,
   _in HV_VP_INDEX            VpIndex,
   _in HV_VTL                 TargetVtl,
   _in HV_INITIAL_VP_CONTEXT  VpVtlContext
);
```

Input Structures

```
typedef struct
{
    UINT64 Rip;
    UINT64 Rsp;
    UINT64 Rflags;

    // Segment selector registers together with their hidden state.
    HV_X64_SEGMENT_REGISTER Cs;
    HV_X64_SEGMENT_REGISTER Ds;
    HV_X64_SEGMENT_REGISTER Es;
    HV_X64_SEGMENT_REGISTER Fs;
    HV_X64_SEGMENT_REGISTER Gs;
    HV_X64_SEGMENT_REGISTER Ss;
    HV_X64_SEGMENT_REGISTER Tr;
    HV_X64_SEGMENT_REGISTER Ldtr;

    // Global and Interrupt Descriptor tables
    HV_X64_TABLE_REGISTER Idtr;
    HV_X64_TABLE_REGISTER Gdtr;

    // Control registers and MSR's
    UINT64 Efer;
    UINT64 Cr0;
    UINT64 Cr3;
    UINT64 Cr4;
    UINT64 MsrCrPat;

} HV_INITIAL_VP_CONTEXT;
```

Native Interface

HvEnableVpVtl			
	Call Code = 0x000f		
➡ Input Parameters			
0	TargetPartitionId (8 bytes)		
8	VpIndex (4 bytes)	TargetVtl (1 byte)	RsvdZ (3 bytes)
16	Rip (8 bytes)		
24	Rsp (8 bytes)		
32	Rflags (8 bytes)		
40	Cs[0] (8 bytes)		
48	Cs[1] (8 bytes)		
56	Ds[0] (8 bytes)		
64	Ds[1] (8 bytes)		
72	Es[0] (8 bytes)		
80	Es[1] (8 bytes)		
88	Fs[0] (8 bytes)		
96	Fs[1] (8 bytes)		
104	Gs[0] (8 bytes)		
112	Gs[1] (8 bytes)		
120	Ss[0] (8 bytes)		
128	Ss[1] (8 bytes)		
136	Ts[0] (8 bytes)		
144	Ts[1] (8 bytes)		
152	Ltdr[0] (8 bytes)		
160	Ltdr[1] (8 bytes)		
168	Itdr[0] (8 bytes)		

176	Itdr[1] (8 bytes)
184	Gtdr[0] (8 bytes)
192	Gtdr[1] (8 bytes)
200	Efer (8 bytes)
208	Cr0 (8 bytes)
216	Cr3 (8 bytes)
224	Cr4 (8 bytes)
232	MsrCrPat (8 bytes)

Input Parameters

*TargetPartitionId s*upplies the partition ID of the partition this request is for.

VpIndex specifies the index of the virtual processor on which to enable the VTL.

TargetVtl specifies the VTL to be enabled by this hypercall.

VpVtlContext gives the initial context in which the VP should start upon the first entry to the target VTL.

Output Parameters

None.

Restrictions

In general, a VTL can only be enabled by a higher VTL. There is one exception to this rule: the highest VTL enabled for a partition can enable a higher target VTL. These restrictions are only enforced for cases where a partition is launching a VTL for itself. If the host partition is enabling a VTL for a child partition, these restrictions do not apply.

Once the target VTL is enabled on a VP, all other calls to enable the VTL must come from equal or greater VTLs.

This hypercall will fail if called to enable a VTL that is already enabled for a VP.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller does not possess the CpuManagement privilege.
HV_STATUS_INVALID_PARTITION_ID	A partition with the specified partition Id does not exist.
HV_STATUS_INVALID_PARAMETER	The hypervisor could not perform the operation because an invalid parameter was specified.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition's state was not appropriate for the requested operation.
HV_STATUS_INVALID_VP_STATE	A virtual processor is not in the correct state for the performance of the indicated operation.
HV_STATUS_INVALID_VTL_STATE	The supplied virtual trust level is not in the correct state to perform the requested operation.
HV_STATUS_INVALID_REGISTER_VALUE	The supplied register value is invalid.
HV_STATUS_INVALID_VP_INDEX	The specified VP index is invalid.

15.15.5 **HvVtlCall**

HvVtlCall switches into the next highest VTL enabled on the VP. For more details, see 15.6.1.

Wrapper Interface

```
HV_STATUS
HvVtlCall ();
```

Native Interface

HvVtlCall	
	Call Code = 0x0011

Input Parameters

See 15.6.1 for details about the input control value.

Output Parameters

None.

Restrictions

See 15.6.1.1.

Return Values

None.

15.15.6 HvVtlReturn

HvVtlReturn switches into the next lowest VTL enabled on the VP. For more details, see 15.7.1.

Wrapper Interface

```
HV_STATUS
HvVtlReturn ();
```

Native Interface

HvVtlReturn	
	Call Code = 0x0012

Input Parameters

See 15.7 for details about the input control value and “fast return”.

Output Parameters

None.

Restrictions

See 15.7.2

Return Values

None.

16 Nested Virtualization

16.1 Overview

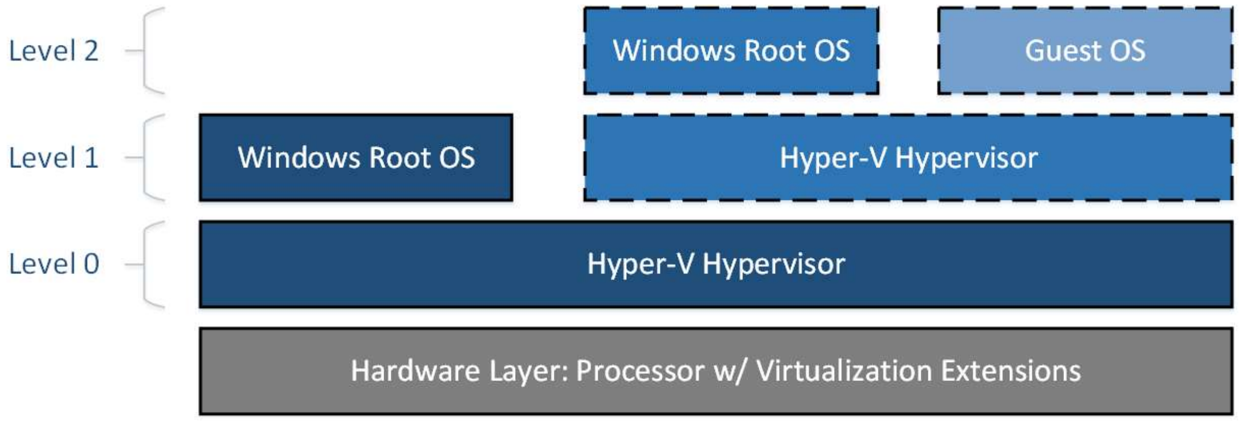
Nested virtualization refers to the Hyper-V hypervisor emulating hardware virtualization extensions. These emulated extensions can be used by other virtualization software (e.g. a nested hypervisor) to run on the Hyper-V platform.

16.2 Definitions

The following terminology is used to define various levels of nested virtualization:

Term	Definition
L0 Hypervisor	The Hyper-V hypervisor running on physical hardware.
L1 Root	The Windows root operating system.
L1 Guest	A Hyper-V virtual machine without a nested hypervisor.
L1 Hypervisor	A nested hypervisor running within a Hyper-V virtual machine.
L2 Root	A root Windows operating system, running within the context of a Hyper-V virtual machine.
L2 Guest	A nested virtual machine, running within the context of a Hyper-V virtual machine.

The diagram below shows how "levels" are used to describe a case where Hyper-V is both the L0 and L1 hypervisor:



16.3 Requirements

16.3.1 Guest Partition

This capability is only available to guest partitions. It must be enabled per virtual machine.

Nested virtualization is not supported in a Windows root partition.

16.3.2 Supported Platforms

Nested virtualization is supported on Intel platforms only.

16.4 Guest Enlightenments

Compared to bare-metal, hypervisors can incur a significant performance regression when running in a VM. L1 hypervisors can be optimized to run in a Hyper-V VM by using enlightened interfaces provided by the L0 hypervisor.

16.4.1 **Enlightened Interface Discovery**

Support for an enlightened VMCS interface is reported with CPUID leaf 0x40000004. If an enlightened VMCS interface is supported, additional nested enlightenments may be discovered by reading the CPUID leaf 0x4000000A (see 2.4.11).

16.5 Enlightened VMCS

On Intel platforms, virtualization software uses virtual machine control data structures (VMCSs) to configure processor behavior related to virtualization. VMCSs must be made active using a VMPTRLD instruction and modified using VMREAD and VMWRITE instructions. These instructions are often a significant bottleneck for nested virtualization because they must be emulated.

The hypervisor exposes an “enlightened VMCS” feature which can be used to control virtualization-related processor behavior using a data structure in guest physical memory. This data structure can be modified using normal memory access instructions, thus there is no need for the L1 hypervisor to execute VMREAD or VMWRITE or VMPTRLD instructions.

The L1 hypervisor may choose to use enlightened VMCSs by writing 1 to the corresponding field in the VP assist page (see section 7.8.7). Another field in the VP assist page controls the currently active enlightened VMCS. Each enlightened VMCS is exactly one page (4 KB) in size and must be initially zeroed. No VMPTRLD instruction must be executed to make an enlightened VMCS active or current.

After the L1 hypervisor performs a VM entry with an enlightened VMCS, the VMCS is considered active on the processor. An enlightened VMCS can only be active on a single processor at the same time. The L1 hypervisor can execute a VMCLEAR instruction to transition an enlightened VMCS from the active to the non-active state. Any VMREAD or VMWRITE instructions while an enlightened VMCS is active is unsupported and can result in unexpected behavior.

The enlightened VMCS type is defined in section 16.11.2. All non-synthetic fields map to an Intel physical VMCS encoding, which is defined in section 16.11.4.

16.5.1 **Enlightened VMCS Versioning**

The enlightened VMCS structure is versioned to account for future changes. Each enlightened VMCS structure contains a version field, which is reported by the L0 hypervisor (see 2.4.11)

The only VMCS version currently supported is 1.

16.5.2 **Clean Fields**

The L0 hypervisor may choose to cache parts of the enlightened VMCS. The enlightened VMCS clean fields control which parts of the enlightened VMCS are reloaded from guest memory on a nested VM entry. The L1 hypervisor must clear the corresponding VMCS clean fields every time it modifies the enlightened VMCS, otherwise the L0 hypervisor might use a stale version.

The clean fields enlightenment is controlled via the synthetic “CleanFields” field of the enlightened VMCS. By default, all bits are set such that the L0 hypervisor must reload the corresponding VMCS fields for each nested VM entry.

16.5.3 **Enlightened MSR Bitmap**

On Intel platforms, the L0 hypervisor emulates the VMX "MSR-Bitmap Address" controls that allow virtualization software to control which MSR accesses generate intercepts.

The L1 hypervisor may collaborate with the L0 hypervisor to make MSR accesses more efficient. It can enable enlightened MSR bitmaps by setting the corresponding field in the enlightened VMCS (see 16.11.2) to 1. When enabled, the L0 hypervisor does not monitor the MSR bitmaps for changes. Instead, the L1 hypervisor must invalidate the corresponding clean field after making changes to one of the MSR bitmaps.

16.6 Compatibility with Live Migration

Hyper-V has the ability to live migrate a child partition from one host to another host. Live migrations are typically transparent to the child partition. However, in the case of nested virtualization, the L1 hypervisor may need to be aware of migrations.

16.6.1 **Live Migration Notifications**

An L1 hypervisor can request to be notified when its partition is migrated. This capability is enumerated in CPUID as "AccessReenlightenmentControls" privilege (see 2.4.10).

The L0 hypervisor exposes a synthetic MSR (`HV_X64_MSR_REENLIGHTENMENT_CONTROL`) that may be used by the L1 hypervisor to configure an interrupt vector and target processor. The L0 hypervisor will inject an interrupt with the specified vector after each migration.

```
#define HV_X64_MSR_REENLIGHTENMENT_CONTROL (0x40000106)

typedef union
{
    UINT64 AsUINT64;
    struct
    {

        UINT64 Vector       :8;
        UINT64 RsvdZ1       :8;
        UINT64 Enabled      :1;
        UINT64 RsvdZ2       :15;
        UINT64 TargetVp     :32;
    };
} HV_REENLIGHTENMENT_CONTROL;
```

The specified vector must correspond to a fixed APIC interrupt. TargetVp specifies the virtual processor index.

16.6.2 **TSC Emulation**

A guest partition may be live migrated between two machines with different TSC frequencies. In those cases, the TscScale value from the reference TSC page (see section 12.6) may need to be recomputed.

The L0 hypervisor optionally emulates all TSC accesses after a migration until the L1 hypervisor has had the opportunity to recompute the TscScale value. The L1 hypervisor can opt into TSC Emulation by writing to the `HV_X64_MSR_TSC_EMULATION_CONTROL` MSR. If opted in, the L0 hypervisor emulates TSC accesses after a migration takes place.

The L1 hypervisor can query if TSC accesses are currently being emulated using the `HV_X64_MSR_TSC_EMULATION_STATUS` MSR. For example, the L1 hypervisor could subscribe to Live

Migration notifications (see 16.6) and query the TSC status after it receives the migration interrupt. It can also turn off TSC emulation (after it updates the TscScale value) using this MSR.

```
#define HV_X64_MSR_TSC_EMULATION_CONTROL    (0x40000107)
#define HV_X64_MSR_TSC_EMULATION_STATUS     (0x40000108)

typedef union {
    UINT64 AsUINT64;
    struct
    {
        UINT64 Enabled      :1;
        UINT64 RsvdZ        :63;
    };

} HV_TSC_EMULATION_CONTROL;

typedef union {
    UINT64 AsUINT64;
    struct
    {
        UINT64 InProgress   : 1;
        UINT64 RsvdP1       : 63;
    };

} HV_TSC_EMULATION_STATUS;
```

16.7 Virtual TLB

The virtual TLB exposed by the hypervisor may be extended to cache translations from L2 GPAs to GPAs. As with the TLB on a logical processor, the virtual TLB is a non-coherent cache, and this non-coherence is visible to guests. The hypervisor exposes operations to manage the TLB.

On Intel platforms, the L0 hypervisor virtualizes the following additional ways to manage the TLB:

- The INVVPID instruction can be used to invalidate cached GVA to GPA or SPA mappings
- The INVEPT instruction can be used to invalidate cached L2 GPA to GPA mappings

16.8 Direct Virtual Flush

The hypervisor exposes hypercalls (HvFlushVirtualAddressSpace, HvFlushVirtualAddressSpaceEx, HvFlushVirtualAddressList, and HvFlushVirtualAddressListEx) that allow operating systems to more efficiently manage the virtual TLB. The L1 hypervisor can choose to allow its guest to use those hypercalls and delegate the responsibility to handle them to the L0 hypervisor. This requires the use of enlightened VMCSs and of a partition assist page.

When enlightened VMCSs are in use, the virtual TLB tags all cached mappings with an identifier of the enlightened VMCS that created them. In response to a direct virtual flush hypercall from a L2 guest, the L0 hypervisor invalidates all cached mappings created by enlightened VMCSs where

- The VmId is the same as the caller’s VmId
- Either the VpId is contained in the specified ProcessorMask or HV_FLUSH_ALL_PROCESSORS is specified

16.8.1 **Configuration**

Direct handling of virtual flush hypercalls is enabled by:

- Setting the NestedEnlightenmentsControl.Features.DirectHypercall field of the virtual processor assist page (see section 7.8.7.1) to 1.

- Setting the EnlightenmentsControl.NestedFlushVirtualHypercall field of an enlightened VMCS to 1.

Before enabling it, the L1 hypervisor must configure the following additional fields of the enlightened VMCS:

- VpId: ID of the virtual processor that the enlightened VMCS controls.
- VmId: ID of the virtual machine that the enlightened VMCS belongs to.
- PartitionAssistPage: Guest physical address of the partition assist page.

The L1 hypervisor must also expose the following capabilities to its guests via CPUID (see 9.1.3):

- UseHypercallForLocalFlush
- UseHypercallForRemoteFlush

16.8.2 **Partition Assist Page**

The partition assist page is a page-size aligned page-size region of memory that the L1 hypervisor must allocate and zero before direct flush hypercalls can be used. Its GPA must be written to the corresponding field in the enlightened VMCS.

```
struct
{
    UINT32 TlbLockCount;
} VM_PARTITION_ASSIST_PAGE;
```

16.8.3 **Synthetic VM-Exit**

If the TlbLockCount of the caller's partition assist page is non-zero, the L0 hypervisor delivers a VM-Exit with a synthetic exit reason to the L1 hypervisor after handling a direct virtual flush hypercall.

```
#define HV_VMX_SYNTHETIC_EXIT_REASON_TRAP_AFTER_FLUSH  0x10000031
```

16.9 Second Level Address Translation

When nested virtualization is enabled for a guest partition, the memory management unit (MMU) exposed by the partition includes support for second level address translation. Second level address translation is a capability that can be used by the L1 hypervisor to virtualize physical memory. When in use, certain addresses that would be treated as guest physical addresses (GPAs) are treated as L2 guest physical addresses (L2 GPAs) and translated to GPAs by traversing a set of paging structures.

The L1 hypervisor can decide how and where to use second level address spaces. Each second level address space is identified by a guest defined 64-bit ID value. On Intel platforms, this value is the same as the address of the EPT PML4 table.

16.9.1 **Compatibility**

On Intel platforms, the second level address translation capability exposed by the hypervisor is generally compatible with VMX support for address translation. However, the following guest-observable differences exist:

- Internally, the hypervisor may use shadow page tables that translate L2 GPAs to SPAs. In such implementations, these shadow page tables appear to software as large TLBs. However, several differences may be observable. First, shadow page tables can be shared between two virtual processors, whereas traditional TLBs are per-processor structures and are independent. This sharing may be visible because a page access by one virtual processor can fill a shadow page table entry that is subsequently used by another virtual processor.

- Some hypervisor implementations may use internal write protection of guest page tables to lazily flush MMU mappings from internal data structures (for example, shadow page tables). This is architecturally invisible to the guest because writes to these tables will be handled transparently by the hypervisor. However, writes performed to the underlying GPA pages by other partitions or by devices may not trigger the appropriate TLB flush.
- On some hypervisor implementations, a second level page fault (“EPT violation”) might not invalidate cached mappings.

16.10 Nested MSR Access Restriction

With VSM enabled in the root partition, the Hyper-V hypervisor filters MSR access for security purposes (guest partition MSR access is already filtered or virtualized). For example, access to non-architectural or vendor-specific MSRs are blocked. This protection also applies in the nested virtualization case. When the Hyper-V hypervisor is running nested within a virtual machine, and the L0 hypervisor chooses to reflect an MSR access to the L1 Hyper-V hypervisor, the access may be dropped.

To avoid the MSR access being filtered out, the L0 hypervisor may choose not to reflect the MSR access to the L1 hypervisor and handle it directly. In this case, caution should be taken to avoid a “confused deputy” attack in which the L0 hypervisor is used to work around VSM protections or attack a higher privileged context.

16.11 Nested Virtualization Data Types

16.11.1 GPA Range

```
typedef union
{
    UINT64 AsUINT64;

    struct
    {
        UINT64 AdditionalPages : 11;
        UINT64 LargePage : 1;
        UINT64 BasePfn : 52;

    };

    struct
    {
        UINT64 : 12;
        UINT64 PageSize : 1;
        UINT64 Reserved : 8;
        UINT64 BaseLargePfn : 43;
    };

} HV_GPA_PAGE_RANGE;
```

16.11.2 Enlightened VMCS

```
Below is the type definition for the enlightened VMCS. The corresponding
Intel physical VMCS encoding for each field can be found in 16.11.4. Note
that some enlightened VMCS fields are synthetic, and therefore will not have
a corresponding physical VMCS encoding.

typedef struct
{
    UINT32  VersionNumber;
    UINT32  AbortIndicator;

    UINT16 HostEsSelector;
    UINT16 HostCsSelector;
    UINT16 HostSsSelector;
    UINT16 HostDsSelector;
    UINT16 HostFsSelector;
    UINT16 HostGsSelector;
    UINT16 HostTrSelector;
    UINT64 HostPat;
    UINT64 HostEfer;
    UINT64 HostCr0;
    UINT64 HostCr3;
    UINT64 HostCr4;
    UINT64 HostSysenterEspMsr;
    UINT64 HostSysenterEipMsr;
    UINT64 HostRip;
    UINT32 HostSysenterCsMsr;
    UINT32 PinControls;
    UINT32 ExitControls;
    UINT32 SecondaryProcessorControls;
    HV_GPA IoBitmapA;
    HV_GPA IoBitmapB;
    HV_GPA MsrBitmap;
    UINT16 GuestEsSelector;
    UINT16 GuestCsSelector;
    UINT16 GuestSsSelector;
    UINT16 GuestDsSelector;
    UINT16 GuestFsSelector;
    UINT16 GuestGsSelector;
    UINT16 GuestLdtrSelector;
    UINT16 GuestTrSelector;
    UINT32 GuestEsLimit;
    UINT32 GuestCsLimit;
    UINT32 GuestSsLimit;
    UINT32 GuestDsLimit;
    UINT32 GuestFsLimit;
    UINT32 GuestGsLimit;
    UINT32 GuestLdtrLimit;
    UINT32 GuestTrLimit;
    UINT32 GuestGdtrLimit;
    UINT32 GuestIdtrLimit;
    UINT32 GuestEsAttributes;
    UINT32 GuestCsAttributes;
    UINT32 GuestSsAttributes;
    UINT32 GuestDsAttributes;
    UINT32 GuestFsAttributes;
    UINT32 GuestGsAttributes;
    UINT32 GuestLdtrAttributes;
    UINT32 GuestTrAttributes;
    UINT64 GuestEsBase;
    UINT64 GuestCsBase;
    UINT64 GuestSsBase;
    UINT64 GuestDsBase;
    UINT64 GuestFsBase;
    UINT64 GuestGsBase;
    UINT64 GuestLdtrBase;
    UINT64 GuestTrBase;
    UINT64 GuestGdtrBase;
    UINT64 GuestIdtrBase;
    UINT64 Rsvd1[3];
    HV_GPA ExitMsrStoreAddress;
    HV_GPA ExitMsrLoadAddress;
    HV_GPA EntryMsrLoadAddress;
    UINT64 Cr3Target0;
    UINT64 Cr3Target1;
    UINT64 Cr3Target2;
    UINT64 Cr3Target3;
    UINT32 PfecMask;
    UINT32 PfecMatch;
    UINT32 Cr3TargetCount;
    UINT32 ExitMsrStoreCount;
    UINT32 ExitMsrLoadCount;
    UINT32 EntryMsrLoadCount;
    UINT64 TscOffset;
    HV_GPA VirtualApicPage;
    HV_GPA GuestWorkingVmcsPtr;
    UINT64 GuestIa32DebugCtl;
    UINT64 GuestPat;
    UINT64 GuestEfer;
    UINT64 GuestPdpte0;
    UINT64 GuestPdpte1;
    UINT64 GuestPdpte2;
    UINT64 GuestPdpte3;
    UINT64 GuestPendingDebugExceptions;
    UINT64 GuestSysenterEspMsr;
    UINT64 GuestSysenterEipMsr;
    UINT32 GuestSleepState;
    UINT32 GuestSysenterCsMsr;
    UINT64 Cr0GuestHostMask;
    UINT64 Cr4GuestHostMask;
    UINT64 Cr0ReadShadow;
    UINT64 Cr4ReadShadow;
    UINT64 GuestCr0;
    UINT64 GuestCr3;
    UINT64 GuestCr4;
    UINT64 GuestDr7;
    UINT64 HostFsBase;
    UINT64 HostGsBase;
    UINT64 HostTrBase;
    UINT64 HostGdtrBase;
    UINT64 HostIdtrBase;
    UINT64 HostRsp;
    UINT64 EptRoot;
    UINT16 Vpid;
    UINT16 Rsvd2[3];
    UINT64 Rsvd3[5];
    UINT64 ExitEptFaultGpa;
    UINT32 ExitInstructionError;
    UINT32 ExitReason;
    UINT32 ExitInterruptionInfo;
    UINT32 ExitExceptionErrorCode;
    UINT32 ExitIdtVectoringInfo;
    UINT32 ExitIdtVectoringErrorCode;
    UINT32 ExitInstructionLength;
    UINT32 ExitInstructionInfo;
    UINT64 ExitQualification;
    UINT64 ExitIoInstructionEcx;
    UINT64 ExitIoInstructionEsi;
    UINT64 ExitIoInstructionEdi;
    UINT64 ExitIoInstructionEip;
    UINT64 GuestLinearAddress;
    UINT64 GuestRsp;
    UINT64 GuestRflags;
    UINT32 GuestInterruptibility;
    UINT32 ProcessorControls;
    UINT32 ExceptionBitmap;
    UINT32 EntryControls;
    UINT32 EntryInterruptInfo;
    UINT32 EntryExceptionErrorCode;
    UINT32 EntryInstructionLength;
    UINT32 TprThreshold;
    UINT64 GuestRip;

    UINT32 CleanFields;
    UINT32 Rsvd4;
    UINT32 SyntheticControls;
    union
    {
        UINT32 AsUINT32;

        struct
        {
            UINT32 NestedFlushVirtualHypercall : 1;
            UINT32 MsrBitmap : 1;
            UINT32 Reserved : 30;
        };

    } EnlightenmentsControl;

    UINT32 VpId;
    UINT64 VmId;
    UINT64 PartitionAssistPage;
    UINT64 Rsvd5[4];

    UINT64 GuestBndcfgs;
    UINT64 Rsvd6[7];
    UINT64 XssExitingBitmap;
    UINT64 EnclsExitingBitmap;
    UINT64 Rsvd7[6];

} HV_VMX_ENLIGHTENED_VMCS;
```

16.11.3 **Clean Fields**

```
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_NONE            (0)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_IO_BITMAP       (1 << 0)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_MSR_BITMAP      (1 << 1)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP2    (1 << 2)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP1    (1 << 3)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_PROC    (1 << 4)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_EVENT   (1 << 5)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_ENTRY   (1 << 6)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_EXCPN   (1 << 7)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR            (1 << 8)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_XLAT    (1 << 9)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_BASIC     (1 << 10)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1      (1 << 11)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2      (1 << 12)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER    (1 << 13)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1       (1 << 14)
#define HV_VMX_ENLIGHTENED_CLEAN_FIELD_ENLIGHTENMENTSCONTROL (1 << 15)
```

16.11.4 **Physical VMCS Encoding**

The following table maps the Intel physical VMCS encoding to its corresponding enlightened VMCS field name, as well as its corresponding clean field name.

VMCS Encoding	Enlightened Name	Field Size	Clean Field Name
0x0000681e	GuestRip	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_NONE
0x0000401c	TprThreshold	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_NONE
0x0000681c	GuestRsp	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_BASIC
0x00006820	GuestRflags	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_BASIC
0x00004824	GuestInterruptibility	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_BASIC
0x00004002	ProcessorControls	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_PROC
0x00004004	ExceptionBitmap	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_EXCPN
0x00004012	EntryControls	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_ENTRY
0x00004016	EntryInterruptInfo	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_EVENT
0x00004018	EntryExceptionErrorCode	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_EVENT
0x0000401a	EntryInstructionLength	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_EVENT
0x00000c00	HostEsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00000c02	HostCsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00000c04	HostSsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00000c06	HostDsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00000c08	HostFsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1

0x00000c0a	HostGsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00000c0c	HostTrSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00002c00	HostPat	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00002c02	HostEfer	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00006c00	HostCr0	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00006c02	HostCr3	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00006c04	HostCr4	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00006c10	HostSysenterEspMsr	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00006c12	HostSysenterEipMsr	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00006c16	HostRip	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00004c00	HostSysenterCsMsr	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_GRP1
0x00004000	PinControls	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP1
0x0000400c	ExitControls	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP1
0x0000401e	SecondaryProcessorControls	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP1
0x00002000	IoBitmapA	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_IO_BITMAP
0x00002002	IoBitmapB	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_IO_BITMAP
0x00002004	MsrBitmap	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_MSR_BITMAP
0x00000800	GuestEsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00000802	GuestCsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00000804	GuestSsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00000806	GuestDsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00000808	GuestFsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000080a	GuestGsSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000080c	GuestLdtrSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000080e	GuestTrSelector	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004800	GuestEsLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004802	GuestCsLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004804	GuestSsLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004806	GuestDsLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004808	GuestFsLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000480a	GuestGsLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000480c	GuestLdtrLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000480e	GuestTrLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004810	GuestGdtrLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004812	GuestIdtrLimit	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004814	GuestEsAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2

0x00004816	GuestCsAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004818	GuestSsAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000481a	GuestDsAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000481c	GuestFsAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000481e	GuestGsAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004820	GuestLdtrAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00004822	GuestTrAttributes	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006806	GuestEsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006808	GuestCsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000680a	GuestSsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000680c	GuestDsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x0000680e	GuestFsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006810	GuestGsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006812	GuestLdtrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006814	GuestTrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006816	GuestGdtrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00006818	GuestIdtrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP2
0x00002010	TscOffset	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP2
0x00002012	VirtualApicPage	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP2
0x00002800	GuestWorkingVmcsPtr	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00002802	GuestIa32DebugCtl	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00002804	GuestPat	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00002806	GuestEfer	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x0000280a	GuestPdpte0	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x0000280c	GuestPdpte1	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x0000280e	GuestPdpte2	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00002810	GuestPdpte3	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00006822	GuestPendingDebugExceptions	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00006824	GuestSysenterEspMsr	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00006826	GuestSysenterEipMsr	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00004826	GuestSleepState	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x0000482a	GuestSysenterCsMsr	4	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x00006000	Cr0GuestHostMask	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x00006002	Cr4GuestHostMask	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x00006004	Cr0ReadShadow	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x00006006	Cr4ReadShadow	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR

0x00006800	GuestCr0	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x00006802	GuestCr3	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x00006804	GuestCr4	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x0000681a	GuestDr7	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CRDR
0x00006c06	HostFsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER
0x00006c08	HostGsBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER
0x00006c0a	HostTrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER
0x00006c0c	HostGdtrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER
0x00006c0e	HostIdtrBase	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER
0x00006c14	HostRsp	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_HOST_POINTER
0x00000000	Vpid	2	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_XLAT
0x0000201a	EptRoot	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_XLAT
0x00002812	GuestBndcfgs	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_GUEST_GRP1
0x0000202c	XssExitingBitmap	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP2
0x0000202e	EnclsExitingBitmap	8	HV_VMX_ENLIGHTENED_CLEAN_FIELD_CONTROL_GRP2
0x00002400	ExitEptFaultGpa	8	Read only (no corresponding clean field)
0x00004400	ExitInstructionError	4	Read only (no corresponding clean field)
0x00004402	ExitReason	4	Read only (no corresponding clean field)
0x00004404	ExitInterruptionInfo	4	Read only (no corresponding clean field)
0x00004406	ExitExceptionErrorCode	4	Read only (no corresponding clean field)
0x00004408	ExitIdtVectoringInfo	4	Read only (no corresponding clean field)
0x0000440a	ExitIdtVectoringErrorCode	4	Read only (no corresponding clean field)
0x0000440c	ExitInstructionLength	4	Read only (no corresponding clean field)
0x0000440e	ExitInstructionInfo	4	Read only (no corresponding clean field)
0x00006400	ExitQualification	8	Read only (no corresponding clean field)
0x00006402	ExitIoInstructionEcx	8	Read only (no corresponding clean field)
0x00006404	ExitIoInstructionEsi	8	Read only (no corresponding clean field)
0x00006406	ExitIoInstructionEdi	8	Read only (no corresponding clean field)
0x00006408	ExitIoInstructionEip	8	Read only (no corresponding clean field)
0x0000640a	GuestLinearAddress	8	Read only (no corresponding clean field)

16.12 Nested Virtualization Exceptions

The L1 hypervisor can opt in to combining virtualization exceptions in the page fault exception class. The L0 hypervisor advertises support for this enlightment in the Hypervisor Nested Virtualization Features CPUID leaf (reference section 2.4.11).

This enlightenment can be enabled by setting VirtualizationException to “1” in HV_NESTED_ENLIGHTENMENTS_CONTROL datastructure in the Virtual Processor Assist Page (reference section 7.8.7).

16.13 Nested Virtualization Interfaces

16.13.1 **HvFlushGuestPhysicalAddressSpace**

The HvFlushGuestPhysicalAddressSpace hypercall invalidates cached L2 GPA to GPA mappings within a second level address space.

Wrapper Interface

```
HV_STATUS
HVFlushGuestPhysicalAddressSpace (
   __in HV_SPA    AddressSpace
   __in UINT64    Flags
   );
```

Native Interface

HvFlushGuestPhysicalAddressSpace	
	Call Code = 0x00AF
➡ Input Parameters	
0	AddressSpace (8 bytes)
8	Flags (8 bytes)

Description

The virtual TLB invalidation operation acts on all processors.

On Intel platforms, the HvFlushGuestPhysicalAddressSpace hypercall is like the execution of an INVEPT instruction with type “single-context” on all processors.

All flags are reserved and must be set to zero.

This call guarantees that by the time control returns to the caller, the observable effects of all flushes have occurred.

If the TLB is currently “locked”, the caller’s virtual processor is suspended.

Input Parameters

AddressSpace specifies an address space ID (an EPT PML4 table pointer)

Flags reserved.

Output Parameters

None.

Restrictions

None.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	A parameter is invalid.
HV_STATUS_INVALID_VP_STATE	Occurs when nested virtualization is not enabled, or the VP is currently not in nested hypervisor mode.

16.13.2 **HvFlushGuestPhysicalAddressList**

The HvFlushGuestPhysicalAddressSpace hypercall invalidates cached GVA / L2 GPA to GPA mappings within a portion of a second level address space.

Wrapper Interface

```
HV_STATUS
HVFlushGuestPhysicalAddressList (
   __in HV_SPA                AddressSpace
   __in UINT64                Flags
   __in HV_GPA_PAGE_RANGE     GpaRangeList[]
   );
```

Native Interface

HvFlushGuestPhysicalAddressSpaceList [rep]	
	Call Code = 0x00B0
➡ Input Parameters	
0	AddressSpace (8 bytes)
8	Flags (8 bytes)
➡ Input List Element	
	GpaRangeList (8 bytes)

Description

The virtual TLB invalidation operation acts on all processors.

All flags are reserved and must be set to zero.

This call guarantees that by the time control returns to the caller, the observable effects of all flushes have occurred.

This call takes a list of L2 GPA ranges to flush. Each range has a base L2 GPA. Because flushes are performed with page granularity, the bottom 12 bits of the L2 GPA can be used to define a range length. These bits encode the number of additional pages (beyond the initial page) within the range. This allows each entry to encode a range of 1 to 4096 pages.

If the TLB is currently “locked”, the caller’s virtual processor is suspended.

Input Parameters

AddressSpace specifies and address space ID (an EPT PML4 table pointer)

Flags reserved.

GpaRange specifies an L2 guest physical address range to flush.

Output Parameters

None.

Restrictions

None.

Return Values

Status code	Error condition
HV_STATUS_INVALID_PARAMETER	A parameter is invalid.
HV_STATUS_INVALID_VP_STATE	Occurs when nested virtualization is not enabled, or the VP is currently not in nested hypervisor mode.

16.14 Nested Virtualization MSRs

16.14.1 Nested VP Index Register

MSR address	Register name	Function
0x40001002	HV_X64_MSR_NESTED_VP_INDEX	In a nested root partition, reports the current processor's underlying VP index.

Bits	Description	Attributes
63:0	The current processor's underlying VP index.	Read

16.14.2 Nested SynIC MSRs

In a nested root partition, the following MSRs allow access the corresponding MSRs of the base hypervisor. For detailed information on the behavior of the corresponding registers, reference section 11.8.

To find the index of the underlying processor, callers should first use HV_X64_MSR_NESTED_VP_INDEX.

MSR Address	Register Name	Function
0x40001080	HV_X64_MSR_NESTED_SCONTROL	Used to control specific behaviors of the synthetic interrupt controller of the base hypervisor in a nested root partition.
0x40001081	HV_X64_MSR_NESTED_SVERSION	Specifies the SynIC version of the base hypervisor in a nested root partition.
0x40001082	HV_X64_MSR_NESTED_SIEFP	Controls the base address of the synthetic interrupt event flag page of the base hypervisor in a nested root partition.
0x40001083	HV_X64_MSR_NESTED_SIMP	Controls the base address of the synthetic interrupt parameter page of the base hypervisor in a nested root partition.
0x40001084	HV_X64_MSR_NESTED_EOM	Indicates the end of message in the SynIC of the base hypervisor in a nested root partition.
0x40001090	HV_X64_MSR_NESTED_SINT0	Configures synthetic interrupt source 0 of the base hypervisor in a nested root partition.
0x40001091	HV_X64_MSR_NESTED_SINT1	Configures synthetic interrupt source 1 of the base hypervisor in a nested root partition.
0x40001092	HV_X64_MSR_NESTED_SINT2	Configures synthetic interrupt source 2 of the base hypervisor in a nested root partition.
0x40001093	HV_X64_MSR_NESTED_SINT3	Configures synthetic interrupt source 3 of the base hypervisor in a nested root partition.

MSR Address	Register Name	Function
0x40001094	HV_X64_MSR_NESTED_SINT4	Configures synthetic interrupt source 4 of the base hypervisor in a nested root partition.
0x40001095	HV_X64_MSR_NESTED_SINT5	Configures synthetic interrupt source 5 of the base hypervisor in a nested root partition.
0x40001096	HV_X64_MSR_NESTED_SINT6	Configures synthetic interrupt source 6 of the base hypervisor in a nested root partition.
0x40001097	HV_X64_MSR_NESTED_SINT7	Configures synthetic interrupt source 7 of the base hypervisor in a nested root partition.
0x40001098	HV_X64_MSR_NESTED_SINT8	Configures synthetic interrupt source 8 of the base hypervisor in a nested root partition.
0x40001099	HV_X64_MSR_NESTED_SINT9	Configures synthetic interrupt source 9 of the base hypervisor in a nested root partition.
0x4000109A	HV_X64_MSR_NESTED_SINT10	Configures synthetic interrupt source 10 of the base hypervisor in a nested root partition.
0x4000109B	HV_X64_MSR_NESTED_SINT11	Configures synthetic interrupt source 11 of the base hypervisor in a nested root partition.
0x4000109C	HV_X64_MSR_NESTED_SINT12	Configures synthetic interrupt source 12 of the base hypervisor in a nested root partition.
0x4000109D	HV_X64_MSR_NESTED_SINT13	Configures synthetic interrupt source 13 of the base hypervisor in a nested root partition.

MSR Address	Register Name	Function
0x4000109E	HV_X64_MSR_NESTED_SINT14	Configures synthetic interrupt source 14 of the base hypervisor in a nested root partition.
0x4000109F	HV_X64_MSR_NESTED_SINT15	Configures synthetic interrupt source 15 of the base hypervisor in a nested root partition.

16.14.2.1 **NESTED_SCONTROL Register**

63:1	0
RsvdP	Enable

This register is used to control SynIC behavior of the base hypervisor in a nested root partition.

Bits	Description	Attributes
63:1	RsvdP (value must be preserved)	Read/write
0	Enable When set, this virtual processor will allow message queuing and event flag notifications to be posted to its SynIC. When clear, message queuing and event flag notifications cannot be directed to this virtual processor.	Read/write

16.14.2.2 **NESTED_SVERSION Register**

63:32	31:0
Rsvd	SynIC Version (0x00000001)

16.14.2.3 **NESTED_SIEFP Register**

63:12	11:1	0
SIEFP Base Address	RsvdP	Enable

Bits	Description	Attributes
63:12	Base address (in GPA space) of SIEFP (low 12 bits assumed to be zero)	Read/write

Bits	Description	Attributes
11:1	RsvdP (value should be preserved)	Read/write
0	SIEFP enable	Read/write

16.14.2.4 NESTED_SIMP Register

63:12	11:1	0
SIMP Base Address	RsvdP	Enable

Bits	Description	Attributes
63:12	Base address (in GPA space) of SIMP (low 12 bits assumed to be zero)	Read/write
11:1	RsvdP (value should be preserved)	Read/write
0	SIMP enable	Read/write

16.14.2.5 NESTED_SINTx Registers

63:19	18	17	16	15:8	7:0
RsvdP	Polling	AutoEOI	Mask	RsvdP	Vector

Bits	Description	Attributes
63:19	RsvdP (value should be preserved)	Read/write
18	Polling	Read/write
17	AutoEOI Set if an implicit EOI should be performed upon interrupt delivery	Read/write
16	Set if the SINT is masked	Read/write
15:8	RsvdP (value should be preserved)	Read/write
7:0	Vector	Read/write

16.14.2.6 NESTED_EOM Register

63:0
RsvdZ

Bits	Description	Attributes
63:0	RsvdZ (value should be set to zero)	Write-only trigger

17 Appendix A: Hypercall Code Reference

Note: Previous versions of the TLFS included appendices detailing (1) architectural features exposed via CPUID, (2) architectural MSRs, and (3) vendor specific MSRs. These appendices have been removed. For current information on these features and MSRs, please reference the latest Intel and AMD architectural manuals.

The following is a table of all hypercalls by call code.

Call Code	Rep Call	Fast Call	Hypercall	Caller	Partition Privilege Required (if any)
0x0001		✓	HvSwitchVirtualAddressSpace	Any	UseHypercallForAddressSpaceSwitch
0x0002			HvFlushVirtualAddressSpace	Any	UseHypercallFor[Local][Remote] Flush
0x0003	✓		HvFlushVirtualAddressList	Any	UseHypercallFor[Local][Remote] Flush
0x0004			HvGetLogicalProcessorRunTime	Any	CpuManagement
0x0005 through 0x0007			Reserved	--	
0x0008		✓	HvCallNotifyLongSpinWait	Any	UseHypercallForLongSpinWait
0x00090			HvCallParkedVirtualProcessors	Any	CpuManagement
0x000b			HvCallSendSyntheticClusterIpi	Any	
0x000c	✓		HvCallModifyVtlProtectionMask	Any	
0x000d			HvCallEnablePartitionVtl	Any	
0x000e			HvCallDisablePartitionVtl	Any	
0x000f			HvCallEnableVpVtl	Any	
0x0010			HvCallDisableVpVtl	Any	
0x0011			HvCallVtlCall	Any	
0x0012			HvCallVtlReturn	Any	
0x0013			HvCallFlushVirtualAddressSpaceEx	Any	
0x0014			HvCallFlushVirtualAddressListEx	Any	
0x0015			HvCallSendSyntheticClusterIpiEx	Any	
0x0016 through 0x003F			Reserved	--	
0x0040			HvCreatePartition	Any	CreatePartitions

Call Code	Rep Call	Fast Call	Hypercall	Caller	Partition Privilege Required (if any)
0x0041		✓	HvInitializePartition	Parent	
0x0042		✓	HvFinalizePartition	Parent	
0x0043		✓	HvDeletePartition	Parent	
0x0044			HvGetPartitionProperty	Parent / Root	
0x0045			HvSetPartitionProperty	Parent / Root	
0x0046			HvGetPartitionId	Any	AccessPartitionId
0x0047			HvGetNextChildPartition	Parent	
0x0048	✓		HvDepositMemory	Parent / Root	AccessMemoryPool
0x0049	✓		HvWithdrawMemory	Parent / Root	AccessMemoryPool
0x004A			HvGetMemoryBalance	Parent / Root	AccessMemoryPool
0x004B	✓		HvMapGpaPages	Parent / Root	
0x004C	✓		HvUnmapGpaPages	Parent	
0x004D			HvInstallIntercept	Parent	
0x004E			HvCreateVp	Parent	
0x004F		✓	HvDeleteVp	Parent	
0x0050	✓		HvGetVpRegisters	Any	
0x0051	✓		HvSetVpRegisters	Any	
0x0052			HvTranslateVirtualAddress	Any	
0x0053			HvReadGpa	Parent	
0x0054			HvWriteGpa	Parent	
0x0055			Deprecated	Parent	
0x0056		✓	HvClearVirtualInterrupt	Parent	
0x0057			Deprecated	Parent / Root	CreatePort
0x0058		✓	HvDeletePort	Parent / Root	
0x0059			HvConnectPort	Parent / Root	ConnectPort

Call Code	Rep Call	Fast Call	Hypercall	Caller	Partition Privilege Required (if any)
0x005A			HvGetPortProperty	Parent / Root	
0x005B		✓	HvDisconnectPort	Parent / Root	
0x005C			HvPostMessage	Any	PostMessages
0x005D		✓	HvSignalEvent	Any	SignalEvents
0x005E			HvSavePartitionState	Parent	
0x005F			HvRestorePartitionState	Parent	CreatePartitions
0x0060			HvInitializeEventLogBufferGroup	Root	
0x0061		✓	HvFinalizeEventLogBufferGroup	Root	
0x0062		✓	HvCreateEventLogBuffer	Root	
0x0063		✓	HvDeleteEventLogBuffer	Root	
0x0064			HvMapEventLogBuffer	Root	
0x0065		✓	HvUnmapEventLogBuffer	Root	
0x0066		✓	HvSetEventLogGroupSources	Root	
0x0067		✓	HvReleaseEventLogBuffer	Root	
0x0068		✓	HvFlushEventLogBuffer	Root	
0x0069			HvPostDebugData	Any	Debugging
0x006A			HvRetrieveDebugData	Any	Debugging
0x006B		✓	HvResetDebugSession	Any	Debugging
0x006C			HvMapStatsPage	Parent[2]	AccessStats
0x006D			HvUnmapStatsPage	Parent	AccessStats
0x006E	✓		HvCallMapSparseGpaPages	Parent / Root	
0x006F			HvCallSetSystemProperty	Root	ConfigureProfiler
0x0070			HvCallSetPortProperty	Parent / Root	CreatePort
0x0071 thru 0x0075			Reserved		

[2] Only the root partition may map global statistics pages.

Call Code	Rep Call	Fast Call	Hypercall	Caller	Partition Privilege Required (if any)
0x0076			HvCallAddLogicalProcessor	Root	CpuManagement
0x0077			HvCallRemoveLogicalProcessor	Root	CpuManagement
0x0078			HvCallQueryNumaDistance	Root	CpuManagement
0x0079			HvCallSetLogicalProcessorProperty	Root	CpuManagement
0x007A			HvCallGetLogicalProcessorProperty	Root	CpuManagement
0x007B			HvCallGetSystemProperty	Any	CpuManagement
0x007C			HvCallMapDeviceInterrupt	Root	CpuManagement
0x007D			HvCallUnmapDeviceInterrupt	Root	CpuManagement
0x007E			HvCallRetargetDeviceInterrupt	Any	CpuManagement
0x007F			Reserved	Root	CpuManagement
0x0080			HvCallMapDevicePages	Root	CpuManagement
0x0081			HvCallUnmapDevicePages	Root	CpuManagement
0x0082			HvCallAttachDevice	Root	CpuManagement
0x0083			HvCallDetachDevice	Root	CpuManagement
0x0084			HvCallNotifyStandbyTransition	Root	CpuManagement
0x0085			HvCallPrepareForSleep	Root	CpuManagement
0x0086			HvCallPrepareForHibernate	Root	CpuManagement
0x0087			HvCallNotifyPartitionEvent	Root	CpuManagement
0x0088			HvCallGetLogicalProcessorRegisters	Root	CpuManagement
0x0089			HvCallSetLogicalProcessorRegisters	Root	CpuManagement
0x008A			HvCallQueryAssociatedLpsforMca	Root	CpuManagement
0x008B			HvCallNotifyRingEmpty	Root	CpuManagement
0x008C			HvCallInjectSyntheticMachineCheck	Root	CpuManagement
0x008D			HvCallScrubPartition	Root	
0x008E			HvCallCollectLivedump	Root	Debugging
0x008F			HvCallDisableHypervisor	Root	
0x0090			HvCallModifySparseGpaPages	Root	
0x0091			HvCallRegisterInterceptResult	Root	
0x0092			HvCallUnregisterInterceptResult	Root	

Call Code	Rep Call	Fast Call	Hypercall	Caller	Partition Privilege Required (if any)
0x0094			HvCallAssertVirtualInterrupt	Any	
0x0095			HvCallCreatePort	Root	
0x0096			HvCallConnectPort	Root	
0x0097			HvCallGetSpaPageList	Root	
0x0098			Reserved		
0x0099			HvCallStartVirtualProcessor	Any	
0x009A	✓		HvCallGetVpIndexFromApicId	Any	
0x009A through 0x00AE			Reserved		
0x00AF			HvCallFlushGuestPhysicalAddressSpace	Any	
0x00B0	✓		HvCallFlushGuestPhysicalAddressList	Any	

18 Appendix B: Hypercall Status Code Reference

The following is a table of all hypercall return codes.

Note: several status calls have been removed in this version of the Microsoft hypervisor. These status codes were used to indicate which specific processor features in the supplied restore state were unsupported. This version has removed these specific status codes, and uses status code HV_STATUS_PROCESSOR_FEATURE_NOT_SUPPORTED (0x0020) as a general status code in these situations.

Status Code	Status Name	Meaning
0x0000	HV_STATUS_SUCCESS	The operation succeeded.
0x0001		Reserved.
0x0002	HV_STATUS_INVALID_HYPERCALL_CODE	The hypervisor does not support the operation because the specified hypercall code is not supported.
0x0003	HV_STATUS_INVALID_HYPERCALL_INPUT	The rep count was incorrect (for example, a non-zero rep count was passed to a non-rep call or a zero rep count was passed to a rep call) or a reserved bit in the specified hypercall input value was non-zero.
0x0004	HV_STATUS_INVALID_ALIGNMENT	The specified input and/or output GPA pointers were not aligned to 8 bytes or the specified input and/or output parameters lists spanned a page boundary.
0x0005	HV_STATUS_INVALID_PARAMETER	One or more input parameters were invalid.
0x0006	HV_STATUS_ACCESS_DENIED	The caller did not possess sufficient access rights to perform the requested operation.
0x0007	HV_STATUS_INVALID_PARTITION_STATE	The specified partition's state was not appropriate for the requested operation.
0x0008	HV_STATUS_OPERATION_DENIED	The operation could not be performed. (The actual cause depends on the operation.)
0x0009	HV_STATUS_UNKNOWN_PROPERTY	The specified partition property ID is not a recognized property.
0x000A	HV_STATUS_PROPERTY_VALUE_OUT_OF_RANGE	The specified value of a partition property is out of range or violates an invariant.
0x000B	HV_STATUS_INSUFFICIENT_MEMORY	Insufficient memory exists for the call to succeed.
0x000C	HV_STATUS_PARTITION_TOO_DEEP	The maximum partition depth has been exceeded for the partition hierarchy.
0x000D	HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
0x000E	HV_STATUS_INVALID_VP_INDEX	The specified VP index is invalid.
0x000F		Reserved
0x0010		Reserved
0x0011	HV_STATUS_INVALID_PORT_ID	The specified port ID is not unique or does not exist.

Status Code	Status Name	Meaning
0x0012	HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is not unique or does not exist.
0x0033	HV_STATUS_INSUFFICIENT_BUFFERS	The target port does not have sufficient buffers for the caller to post a message.
0x0014	HV_STATUS_NOT_ACKNOWLEDGED	An external interrupt has not previously been asserted and acknowledged by the virtual processor prior to clearing it.
0x0015	HV_STATUS_INVALID_VP_STATE	A virtual processor is not in the correct state for the performance of the indicated operation.
0x0016	HV_STATUS_ACKNOWLEDGED	An external interrupt cannot be asserted because a previously-asserted external interrupt was acknowledged by the virtual processor and has not yet been cleared.
0x0017	HV_STATUS_INVALID_SAVE_RESTORE_STATE	The initial call to HvSavePartitionState or HvRestorePartitionState specifying HV_SAVE_RESTORE_STATE_START was not made at the beginning of the save/restore process.
0x0018	HV_STATUS_INVALID_SYNIC_STATE	The operation could not be performed because a required feature of the SynIC was disabled.
0x0019	HV_STATUS_OBJECT_IN_USE	The operation could not be performed because the object or value was either already in use or being used for a purpose that would not permit it.
0x001A	HV_STATUS_INVALID_PROXIMITY_DOMAIN_INFO	The *Flags* field included an invalid mask value in the proximity domain information.
		The *Id* field contained an invalid ACPI node ID in the proximity domain information.
0x001B	HV_STATUS_NO_DATA	An attempt to retrieve data failed because none was available.
0x001C	HV_STATUS_INACTIVE	The physical connection being used for debugging has not recorded any receive activity since the last operation.
0x001D	HV_STATUS_NO_RESOURCES	A resource is unavailable for allocation. This may indicate that there is a resource shortage or that an implementation limitation may have been reached.
0x001E	HV_STATUS_FEATURE_UNAVAILABLE	A hypervisor feature is not available to the caller.
0x001F	HV_STATUS_PARTIAL_PACKET	The debug packet returned is only a partial packet due to an I/O error.
0x0020	HV_STATUS_PROCESSOR_FEATURE_NOT_SUPPORTED	The supplied restore state requires an unsupported processor feature.
0x0030	HV_STATUS_PROCESSOR_CACHE_LINE_FLUSH_SIZE_INCOMPATIBLE	The processor's cache line flush size is not supported.
0x0033	HV_STATUS_INSUFFICIENT_BUFFER	The specified buffer was too small to contain all of the requested data.
0x0037	HV_STATUS_INCOMPATIBLE_PROCESSOR	The processor architecture is not supported.

Status Code	Status Name	Meaning
0x0038	HV_STATUS_INSUFFICIENT_DEVICE_DOMAINS	The maximum number of domains supported by the platform I/O remapping hardware is currently in use.
0x003C	HV_STATUS_CPUID_FEATURE_VALIDATION_ERROR	Generic logical processor CPUID feature set validation error.
0x003D	HV_STATUS_CPUID_XSAVE_FEATURE_VALIDATION_ERROR	CPUID XSAVE feature validation error.
0x003E	HV_STATUS_PROCESSOR_STARTUP_TIMEOUT	Processor startup timed out.
0x003F	HV_STATUS_SMX_ENABLED	SMX enabled by the BIOS.
0x0041	HV_STATUS_INVALID_LP_INDEX	The hypervisor could not perform the operation because the specified LP index is invalid.
0x0050	HV_STATUS_INVALID_REGISTER_VALUE	The supplied register value is invalid.
0x0055	HV_STATUS_NX_NOT_DETECTED	NX not detected on the machine.
0x0057	HV_STATUS_INVALID_DEVICE_ID	The supplied device ID is invalid.
0x0058	HV_STATUS_INVALID_DEVICE_STATE	The operation is not allowed in the current device state.
0x0059	HV_STATUS_PENDING_PAGE_REQUESTS	The device had pending page requests which were discarded.
0x0060	HV_STATUS_PAGE_REQUEST_INVALID	The supplied page request specifies a memory access that the guest does not have permissions to perform.
0x0071	HV_STATUS_OPERATION_FAILED	The requested operation failed.
0x0072	HV_STATUS_NOT_ALLOWED_WITH_NESTED_VIRT_ACTIVE	The requested operation is not allowed due to one or more virtual processors having nested virtualization active.

19 Appendix C: Hypervisor Synthetic MSRs

Note: Previous versions of the TLFS included appendices detailing (1) architectural features exposed via CPUID, (2) architectural MSRs, and (3) vendor specific MSRs. These appendices have been removed. For current information on these features and MSRs, please reference the latest Intel and AMD architectural manuals.

The following is a table of new MSR values defined by the hypervisor.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x40000000	HV_X64_MSR_GUEST_OS_ID	AccessHypercallMsrs	R/W	Used to identify the guest OS running in the partition. See section 2.6.
0x40000001	HV_X64_MSR_HYPERCALL	AccessHypercallMsrs	R/W	Used to enable the hypercall interface. See section 3.13.
0x40000002	HV_X64_MSR_VP_INDEX	AccessVpIndex	R	Specifies the virtual processor's index. See section 7.8.1.
0x40000003	HV_X64_MSR_RESET	AccessSystemResetMsr	R/W	Used to perform a hypervisor-controlled reboot operation.
0x40000010	HV_X64_MSR_VP_RUNTIME	AccessVpRuntimeMsr	R	Specifies the virtual processor's run time in 100ns units. See section 7.9.1.
0x40000020	HV_X64_MSR_TIME_REF_COUNT	AccessPartitionReferenceCounter	R	Partition-wide reference counter.
0x40000021	HV_X64_MSR_REFERENCE_TSC	AccessPartitionReferenceCounter	R	Partition-wide reference time stamp counter.
0x40000022	HV_X64_MSR_TSC_FREQUENCY	AccessFrequencyRegs	R	Specifies the frequency, in Hz, of the TSC, as reported by the hypervisor.
0x40000023	HV_X64_MSR_APIC_FREQUENCY	AccessFrequencyRegs	R	Specifies the frequency, in Hz, of the local APIC, as reported by the hypervisor.
0x40000040	HV_X64_MSR_NPIEP_CONFIG		R/W	Configures non-privileged instruction execution prevention
0x40000070	HV_X64_MSR_EOI	AccessApicMsrs	W	Fast access to the APIC EOI register. See section 10.2.3.
0x40000071	HV_X64_MSR_ICR	AccessApicMsrs	R/W	Fast access to the APIC ICR high and ICR low registers. See section 10.2.3.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x40000072	HV_X64_MSR_TPR	AccessApicMsrs	R/W	Fast access to the APIC TPR register (use CR8 in 64-bit mode). See section 10.2.3.
0x40000073	HV_X64_MSR_VP_ASSIST_PAGE	AccessApicMsrs	R/W	Enables lazy EOI processing. See section 10.3.4.
0x40000080	HV_X64_MSR_SCONTROL	AccessSynicRegs	R/W	Used to control specific behaviors of the synthetic interrupt controller. See section 11.8.1.
0x40000081	HV_X64_MSR_SVERSION	AccessSynicRegs	R	Specifies the SynIC version.
0x40000082	HV_X64_MSR_SIEFP	AccessSynicRegs	R/W	Controls the base address of the synthetic interrupt event flag page. See section 11.8.3.
0x40000083	HV_X64_MSR_SIMP	AccessSynicRegs	R/W	Controls the base address of the synthetic interrupt parameter page. See section 11.8.4.
0x40000084	HV_X64_MSR_EOM	AccessSynicRegs	W	Indicates the end of message in the SynIC. See section 11.8.6.
0x40000090	HV_X64_MSR_SINT0	AccessSynicRegs	R/W	Configures synthetic interrupt source 0. See section 11.8.5.
0x40000091	HV_X64_MSR_SINT1	AccessSynicRegs	R/W	Configures synthetic interrupt source 1. See section 11.8.5.
0x40000092	HV_X64_MSR_SINT2	AccessSynicRegs	R/W	Configures synthetic interrupt source 2. See section 11.8.5.
0x40000093	HV_X64_MSR_SINT3	AccessSynicRegs	R/W	Configures synthetic interrupt source 3. See section 11.8.5.
0x40000094	HV_X64_MSR_SINT4	AccessSynicRegs	R/W	Configures synthetic interrupt source 4. See section 11.8.5.
0x40000095	HV_X64_MSR_SINT5	AccessSynicRegs	R/W	Configures synthetic interrupt source 5. See section 11.8.5.
0x40000096	HV_X64_MSR_SINT6	AccessSynicRegs	R/W	Configures synthetic interrupt source 6. See section 11.8.5.
0x40000097	HV_X64_MSR_SINT7	AccessSynicRegs	R/W	Configures synthetic interrupt source 7. See section 11.8.5.
0x40000098	HV_X64_MSR_SINT8	AccessSynicRegs	R/W	Configures synthetic interrupt source 8. See section 11.8.5.
0x40000099	HV_X64_MSR_SINT9	AccessSynicRegs	R/W	Configures synthetic interrupt source 9. See section 11.8.5.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x4000009A	HV_X64_MSR_SINT10	AccessSynicRegs	R/W	Configures synthetic interrupt source 10. See section 11.8.5.
0x4000009B	HV_X64_MSR_SINT11	AccessSynicRegs	R/W	Configures synthetic interrupt source 11. See section 11.8.5.
0x4000009C	HV_X64_MSR_SINT12	AccessSynicRegs	R/W	Configures synthetic interrupt source 12. See section 11.8.5.
0x4000009D	HV_X64_MSR_SINT13	AccessSynicRegs	R/W	Configures synthetic interrupt source 13. See section 11.8.5.
0x4000009E	HV_X64_MSR_SINT14	AccessSynicRegs	R/W	Configures synthetic interrupt source 14. See section 11.8.5.
0x4000009F	HV_X64_MSR_SINT15	AccessSynicRegs	R/W	Configures synthetic interrupt source 15. See section 11.8.5.
0x400000B0	HV_X64_MSR_STIMER0_CONFIG	AccessSyntheticTimerRegs	R/W	Configuration register for synthetic timer 0.
0x400000B1	HV_X64_MSR_STIMER0_COUNT	AccessSyntheticTimerRegs	R/W	Expiration time or period for synthetic timer 0.
0x400000B2	HV_X64_MSR_STIMER1_CONFIG	AccessSyntheticTimerRegs	R/W	Configuration register for synthetic timer 1.
0x400000B3	HV_X64_MSR_STIMER1_COUNT	AccessSyntheticTimerRegs	R/W	Expiration time or period for synthetic timer 1.
0x400000B4	HV_X64_MSR_STIMER2_CONFIG	AccessSyntheticTimerRegs	R/W	Configuration register for synthetic timer 2.
0x400000B5	HV_X64_MSR_STIMER2_COUNT	AccessSyntheticTimerRegs	R/W	Expiration time or period for synthetic timer 2.
0x400000B6	HV_X64_MSR_STIMER3_CONFIG	AccessSyntheticTimerRegs	R/W	Configuration register for synthetic timer 3.
0x400000B7	HV_X64_MSR_STIMER3_COUNT	AccessSyntheticTimerRegs	R/W	Expiration time or period for synthetic timer 3.
0x400000F0	HV_X64_MSR_GUEST_IDLE	AccessGuestIdleReg	R	Trigger the guest's transition to the idle power state
0x40000100	HV_X64_MSR_CRASH_P0	GuestCrashMsrsAvailable	R/W	MSR that is preserved with the guest crash enlightenment.
0x40000101	HV_X64_MSR_CRASH_P1	GuestCrashMsrsAvailable	R/W	MSR that is preserved with the guest crash enlightenment.
0x40000102	HV_X64_MSR_CRASH_P2	GuestCrashMsrsAvailable	R/W	MSR that is preserved with the guest crash enlightenment.
0x40000103	HV_X64_MSR_CRASH_P3	GuestCrashMsrsAvailable	R/W	MSR that is preserved with the guest crash enlightenment.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x40000104	HV_X64_MSR_CRASH_P4	GuestCrashMsrsAvailable	R/W	MSR that is preserved with the guest crash enlightenment.
0x40000105	HV_X64_MSR_CRASH_CTL	GuestCrashMsrsAvailable	R/W	Queries and controls the hypervisor’s guest crash capabilities.
0x40000106	HV_X64_MSR_REENLIGHTENMENT_CONTROL	AccessReenlightenmentControls	R/W	Configures reenlightenment controls (see section 16.6).
0x40000107	HV_X64_MSR_TSC_EMULATION_CONTROL	AccessReenlightenmentControls	R/W	Configures TSC emulation control (see section 16.6)
0x40000108	HV_X64_MSR_TSC_EMULATION_STATUS	AccessReenlightenmentControls	R/W	Returns TSC emulation status (see section 16.6).
0x40000114	HV_X64_MSR_STIME_UNHALTED_TIMER_CONFIG	AccessSyntheticTimerRegs	R/W	Configuration register for the time-unhalted timer.
0x40000115	HV_X64_MSR_STIME_UNHALTED_TIMER_COUNT	AccessSyntheticTimerRegs	R/W	Period for the time-unhalted timer.
0x40001002	HV_X64_MSR_NESTED_VP_INDEX	CpuManagement	R	MSR that queries the underlying VP index when running in a nested hypervisor.
0x40001080	HV_X64_MSR_NESTED_SCONTROL	CpuManagement	R/W	Used to control specific behaviors of the synthetic interrupt controller of the base hypervisor in a nested root partition.
0x40001081	HV_X64_MSR_NESTED_SVERSION	CpuManagement	R	Specifies the SynIC version of the base hypervisor in a nested root partition.
0x40001082	HV_X64_MSR_NESTED_SIEFP	CpuManagement	R/W	Controls the base address of the synthetic interrupt event flag page of the base hypervisor in a nested root partition.
0x40001083	HV_X64_MSR_NESTED_SIMP	CpuManagement	R/W	Controls the base address of the synthetic interrupt parameter page of the base hypervisor in a nested root partition.
0x40001084	HV_X64_MSR_NESTED_EOM	CpuManagement	W	Indicates the end of message in the SynIC of the base hypervisor in a nested root partition.
0x40001090	HV_X64_MSR_NESTED_SINT0	CpuManagement	R/W	Configures synthetic interrupt source 0 of the base hypervisor in a nested root partition.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x40001091	HV_X64_MSR_NESTED_SINT1	CpuManagement	R/W	Configures synthetic interrupt source 1 of the base hypervisor in a nested root partition.
0x40001092	HV_X64_MSR_NESTED_SINT2	CpuManagement	R/W	Configures synthetic interrupt source 2 of the base hypervisor in a nested root partition.
0x40001093	HV_X64_MSR_NESTED_SINT3	CpuManagement	R/W	Configures synthetic interrupt source 3 of the base hypervisor in a nested root partition.
0x40001094	HV_X64_MSR_NESTED_SINT4	CpuManagement	R/W	Configures synthetic interrupt source 4 of the base hypervisor in a nested root partition.
0x40001095	HV_X64_MSR_NESTED_SINT5	CpuManagement	R/W	Configures synthetic interrupt source 5 of the base hypervisor in a nested root partition.
0x40001096	HV_X64_MSR_NESTED_SINT6	CpuManagement	R/W	Configures synthetic interrupt source 6 of the base hypervisor in a nested root partition.
0x40001097	HV_X64_MSR_NESTED_SINT7	CpuManagement	R/W	Configures synthetic interrupt source 7 of the base hypervisor in a nested root partition.
0x40001098	HV_X64_MSR_NESTED_SINT8	CpuManagement	R/W	Configures synthetic interrupt source 8 of the base hypervisor in a nested root partition.
0x40001099	HV_X64_MSR_NESTED_SINT9	CpuManagement	R/W	Configures synthetic interrupt source 9 of the base hypervisor in a nested root partition.
0x4000109A	HV_X64_MSR_NESTED_SINT10	CpuManagement	R/W	Configures synthetic interrupt source 10 of the base hypervisor in a nested root partition.
0x4000109B	HV_X64_MSR_NESTED_SINT11	CpuManagement	R/W	Configures synthetic interrupt source 11 of the base hypervisor in a nested root partition.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x4000109C	HV_X64_MSR_NESTED_SINT12	CpuManagement	R/W	Configures synthetic interrupt source 12 of the base hypervisor in a nested root partition.
0x4000109D	HV_X64_MSR_NESTED_SINT13	CpuManagement	R/W	Configures synthetic interrupt source 13 of the base hypervisor in a nested root partition.
0x4000109E	HV_X64_MSR_NESTED_SINT14	CpuManagement	R/W	Configures synthetic interrupt source 14 of the base hypervisor in a nested root partition.
0x4000109F	HV_X64_MSR_NESTED_SINT15	CpuManagement	R/W	Configures synthetic interrupt source 15 of the base hypervisor in a nested root partition.