Hypervisor Top-Level Functional Specification: Windows Server 2008 R2

May 2012: Version 2.0A

Abstract

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

Contents

1 Introduction

This document is the top-level functional specification (TLFS) of the second-generation Microsoft hypervisor. It specifies the externally-visible behavior. 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 compatible hypervisor from scratch.

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 Interface Requirements and Goals

The hypervisor interfaces are designed with the following requirements and goals in mind:

- The amount of time spent in the hypervisor in response to a hypercall should be bounded to 50μs. This restriction requires that operations acting on large lists or ranges must be piece-wise interruptible.
- To the extent possible, the interfaces should be independent of hardware architecture. Although the first-generation hypervisor will run only on x64 processors, it is highly likely that future versions will be ported to other architectures. It may not be possible to implement certain aspects of an architecture in a transparent manner. The x64 implementation exposes architecture-specifics with the handling of the CPUID instruction (the specifics are detailed in this specification).
- The interfaces should provide significant “future proofing” and therefore should not impose short-sighted restrictions on memory size, address space size, processor count, and so on.
- The interfaces should allow a conforming hypervisor implementation to correctly virtualize all aspects of the underlying system architecture. In some cases, the architecture may allow implementations the freedom not to virtualize all aspects correctly, but it should not constrain an implementation in a way that it cannot virtualize all aspects correctly.

1.3 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.

2 Basic Data Types, Concepts and Notation

2.1 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.

2.2 Hypercall Status Code

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

```
typedef UINT16 HV_STATUS;
```

2.3 Memory 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 4 K.

```
#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 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.

2.4 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.

2.5 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.

2.6 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.

3 Feature and Interface Discovery

3.1 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 4.

3.2 Hypervisor Discovery

Before using any hypervisor interfaces, software should first determine whether it's 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.

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.

3.3 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	Hypervisor CPUID leaf range and vendor ID signature.	
	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 that identifies the hypervisor interface.

Leaf	Information Provided	
0x40000001	Hypervisor vendor-neutral interface identification. This determines the semantics of the leaves from 0x40000002 through 0x400000FF	
	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.

3.4 Microsoft Hypervisor CPUID Leaves

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

Leaf	Information Provided	
0x40000000	Hypervisor CPUID leaf range and vendor ID signature.	
	EAX	The maximum input value for hypervisor CPUID information. On Microsoft hypervisors, this will be at least 0x40000005. The vendor ID signature should be used only for reporting and diagnostic purposes.
	EBX	0x7263694D—“Micr”
	ECX	0x666F736F—“osof”
	EDX	0x76482074—“t Hv”
0x40000001	Hypervisor vendor-neutral interface identification. This determines the semantics of the leaves from 0x40000002 through 0x400000FF.	
	EAX	0x31237648—“Hv#1”
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

Hypervisors conforming to the “Hv#1” interface also provide at least the following leaves:

Leaf	Information Provided	
0x40000002	Hypervisor system identity. This value will be zero until the OS identity MSR is set (see section 3.6); after that, they have the following definitions:	
	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

Leaf	Information Provided	
0x40000003	Feature identification. EAX indicates which features are available to the partition based upon the current partition privileges.	
	EAX	Bit 0:VP Runtime (HV_X64_MSR_VP_RUNTIME) available Bit 1: Partition Reference Counter (HV_X64_MSR_TIME_REF_COUNT) available Bit 2: Basic SynIC MSRs (HV_X64_MSR_SCONTROL through HV_X64_MSR_EOM and HV_X64_MSR_SINT0 through HV_X64_MSR_SINT15) available Bit 3: Synthetic Timer MSRs (HV_X64_MSR_STIMER0_CONFIG through HV_X64_MSR_STIMER3_COUNT) available Bit 4: APIC access MSRs (HV_X64_MSR_EOI, HV_X64_MSR_ICR and HV_X64_MSR_TPR) are available Bit 5: Hypercall MSRs (HV_X64_MSR_GUEST_OS_ID and HV_X64_MSR_HYPERCALL) available Bit 6: Access virtual processor index MSR (HV_X64_MSR_VP_INDEX) available Bit 7: Virtual system reset MSR (HV_X64_MSR_RESET) is available. Bit 8: Access statistics pages MSRs (HV_X64_MSR_STATS_PARTITION_RETAIL_PAGE, HV_X64_MSR_STATS_PARTITION_INTERNAL_PAGE, HV_X64_MSR_STATS_VP_RETAIL_PAGE, HV_X64_MSR_STATS_VP_INTERNAL_PAGE) available. Bit 9: Partition Reference TSC MSR (HV_X64_MSR_REFERENCE_TSC) available. Bit 10: Virtual Guest Idle State MSR (HV_X64_MSR_GUEST_IDLE) available. Bits 11-31: Reserved
	Feature identification: EBX indicates which flags were specified at partition creation. The format is the same as the partition creation flag structure defined in section 5.2.4.	
	EBX	Bit 0: CreatePartitions Bit 1: AccessPartitionId Bit 2: AccessMemoryPool Bit 3: AdjustMessageBuffers Bit 4: PostMessages Bit 5: SignalEvents Bit 6: CreatePort Bit 7: ConnectPort Bit 8: AccessStats Bit 9-10: RsvdZ Bit 11: Debugging Bit 12: CpuManagement Bit 13: ConfigureProfiler Bit 14-31: Reserved
	Feature identification. ECX contains power management related information.	
	ECX	Bit 0-3: Maximum Processor Power State. 0 is C0, 1 is C1, 2 is C2, 3 is C3. Bits 8-31: Reserved

Leaf	Information Provided	
0x40000003	Feature identification. EDX indicates which miscellaneous features are available to the partition.	
	EDX	Bit 0: The MWAIT instruction is available (per section 11.5) 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 Bits 6-31: Reserved
0x40000004	Implementation recommendations. Indicates which behaviors the hypervisor recommends the OS implement for optimal performance.	
	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 6-31: Reserved
	EBX	Recommended number of attempts to retry a spinlock failure before notifying the hypervisor about the failures. 0xFFFFFFFF indicates never to retry.
	ECX	Reserved
	EDX	Reserved
0x40000005	Implementation limits. If any value is zero, the hypervisor does not expose the corresponding information; otherwise, they have these meanings:	
	EAX	The maximum number of virtual processors supported.
	EBX	The maximum number of logical processors supported.
	ECX	Reserved
	EDX	Reserved

Hypervisors conforming to the “Hv#1” interface may optionally also provide the following leaves:

Leaf	Information Provided	
0x40000006	Implementation hardware features. Indicates which hardware-specific features have been detected and are currently in use by the hypervisor.	
	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. Bits 4-31: Reserved for future Intel-specific features.
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved for future AMD-specific features.

3.5 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.

3.6 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 10).

This register's value is initially zero. A non-zero value must be written to it before the hypercall code page can be enabled (see section 4.12). If this register is subsequently zeroed, the hypercall code page will be disabled.

The value written to this MSR is ignored by the hypervisor but may be used in future hypervisor implementations to maintain compatibility with existing guest OSs.

```
#define HV_X64_MSR_GUEST_OS_ID 0x40000000
```

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

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

Bits	Description	Attributes
63:48	Vendor ID Indicates the guest OS vendor. A value of 0 is reserved. A value of 1 indicates Microsoft.	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

4 Hypercall Interface

4.1 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.

4.2 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 4.3), the hypercall returns control back to the calling virtual processor after updating the *rep start index* to 20. (See section 4.7 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.

4.3 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.

4.4 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.

4.5 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.

4.6 Alignment Requirements

Callers must specify the 64-bit guest physical address (GPA) of the input and/or output parameters. 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 8.1.3). 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 section 11.6.

For hypercalls that have output parameters, the hypervisor will validate that the partition can be 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.

4.7 Hypercall Inputs

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

Hypercall input value:

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 If this calling convention is used, the Rep fields must be zero.
RsvdZ	15 bits	Must be zero
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

63:60	59:48	47:44	43:32	31:17	16	15:0
RsvdZ (4 bits)	Rep start index (12 bits)	RsvdZ (4 bits)	Rep count (12 bits)	RsvdZ (15 bits)	Fast (1 bit)	Call Code (16 bits)

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.

Some register values are considered volatile and can be modified by a hypercall. This list includes the condition code registers SF, ZF, OF, CF, AF, and PF. It also includes EAX, ECX, and EDX for x86 callers and RCX, RDX, R8, R9, R10, R11, and XMM0 through XMM5 for x64 callers.

RAX (on x64) and EDX:EAX (on x86) are always overwritten with the hypercall result value (discussed in section 4.8).

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.

Some register values are considered volatile and can be modified by a hypercall. This list includes the condition code registers SF, ZF, OF, CF, AF, and PF. It also includes EAX, ECX, EDX, EDI and ESI for x86 callers and RCX, RDX, R8, R9, R10, R11, and XMM0 through XMM5 for x64 callers.

RAX, RDX and R8 (on x64) and EDX:EAX, EBX:ECX and EDI:ESI (on x86) are always overwritten with the hypercall result value and the output parameters, if any (discussed in section 4.8).

4.7.1 Extended Fast Hypercalls

The hypervisor supports the use of *extended 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 extended fast hypercall interface uses six volatile XMM registers to allow the caller to pass an input parameter block up to 112 bytes in size. Availability of the extended fast hypercall interface is indicated by the presence of a flag returned by the CPUID instruction for a hypervisor leaf (see section 3.4). Any attempt to use this interface when the hypervisor does not indicate availability will result in a #UD fault.

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

x64	x86	Contents
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameter Block (Byte 0-7)
R8	EDI:ESI	Input Parameter Block (Byte 8-15)
XMM0	XMM0	Input Parameter Block (Byte 16-31)
XMM1	XMM1	Input Parameter Block (Byte 32-47)

XMM2	XMM2	Input Parameter Block (Byte 48-63)
XMM3	XMM3	Input Parameter Block (Byte 64-79)
XMM4	XMM4	Input Parameter Block (Byte 80-95)
XMM5	XMM5	Input Parameter Block (Byte 96-111)

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.

Some register values are considered volatile and can be modified by a hypercall. This list includes the condition code registers SF, ZF, OF, CF, AF, and PF. It also includes EAX, ECX, EDX, EDI and ESI for x86 callers and RCX, RDX, R8, R9, R10, R11, and XMM0 through XMM5 for x64 callers.

RAX, RDX and R8 (on x64) and EDX:EAX, EBX:ECX and EDI:ESI (on x86) are always overwritten with the hypercall result value and the output parameters, if any (discussed in section 4.8).

4.8 Hypercall Outputs

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

Hypercall output value:

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

63:40	43:32	31:16	15:0
Rsvd (20 bits)	Reps complete (12 bits)	Rsvd (16 bits)	Result (16 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

4.9 Hypercall Details

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		
	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.

4.10 Hypercall Restrictions

Hypercalls may have restrictions associated with them in order 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 5.2.3 for information regarding privilege flags)
- The calling partition must have been created with a particular creation flag (see section 5.2.4 for information regarding creation flags)
- The partition being acted upon must be in a particular state (see section 5.2.5 for information regarding partition states)
- 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 10.1.3 for information regarding virtual processor states).

4.11 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 4.7.1.

4.11.1 Output Parameter Validity on Failed Hypercalls

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.

4.11.2 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.

4.11.3 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.
	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.

4.12 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 an MSR.

```
#define HV_X64_MSR_HYPERCALL 0x40000001
```

63:12	11:1	0
Guest Physical Page Number of Hypercall Page	RsvdP	Enable

Bits	Description	Attributes
63:12	Hypercall GPFN Indicates the **Guest Physical Page Number** of the hypercall page	Read/write
11:1	RsvdP Guest should ignore on reads and preserve on writes	Reserved
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:

The guest reads CPUID leaf 1 and determines whether a hypervisor is present by checking bit 31 of register ECX.
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.
The guest writes its OS identity into the MSR HV_X64_MSR_GUEST_OS_ID if that register is zero.
The guest reads the Hypercall MSR (HV_X64_MSR_HYPERCALL).
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.
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.
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.
The guest creates an executable VA mapping to the hypercall page GPA.
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 ret (0xC3) to return to the caller. As such, the hypercall must be invoked with a valid stack.

5 Partition Management

5.1 Overview

This section defines interfaces that allow guests to create and delete partitions, to enumerate existing partitions and to manipulate their state.

5.2 Partition Management Data Types

5.2.1 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     0x0000000000000000UI64
```

5.2.2 Partition Properties

Partition properties provide a generic way for a parent partition to control aspects of its child partitions. After a partition property is set, its value is constant unless and until it is again modified by a caller possessing sufficient privilege. Properties are identified by a 32-bit code. Property values are each 64 bits in size.

```
typedef UINT64 HV_PARTITION_PROPERTY;
typedef HV_PARTITION_PROPERTY *PHV_PARTITION_PROPERTY;

typedef enum
{
// Privilege properties
HvPartitionPropertyPrivilegeFlags   = 0x00010000,

// Scheduling properties
HvPartitionPropertyCpuReserve = 0x00020001,
HvPartitionPropertyCpuCap     = 0x00020002,
HvPartitionPropertyCpuWeight  = 0x00020003,

// Timer assist properties
HvPartitionPropertyEmulatedTimerPeriod    = 0x00030000,
HvPartitionPropertyEmulatedTimerControl   = 0x00030001,
HvPartitionPropertyPmTimerAssist    = 0x00030002,

// Debugging properties
HvPartitionPropertyDebugChannelId   = 0x00040000,

// Resource properties
```

```
HvPartitionPropertyVirtualTlbPageCount    = 0x00050000,

// Compatibility properties
HvPartitionPropertyProcessorVendor          = 0x00060000,
HvPartitionPropertyProcessorFeatures        = 0x00060001,
HvPartitionPropertyProcessorXsaveFeatures   = 0x00060002,
HvPartitionPropertyProcessorCLFlushSize     = 0x00060003
} HV_PARTITION_PROPERTY_CODE;
```

The following table explains the meaning of each property. The table also lists the default values.

Partition property	Meaning	Default value
HvPartitionPropertyPrivilegeFlags	For details about the privilege flags, see section 5.2.3.	
HvPartitionPropertyCpuReserve	The amount of per-VP CPU time reserved for this partition. For details, see section 18.2.1.	0x0000000000000000
HvPartitionPropertyCpuCap	The maximum amount of per-VP CPU time the partition is allowed to use. For details, see section 18.2.2.	0x0000000000010000
HvPartitionPropertyCpuWeight	The relative weight for the virtual processors in the partition. For details, see section 18.2.3.	0x0000000000000064
HvPartitionPropertyEmulatedTimerPeriod	The periodic timer assist period. For details, see section 15.1.5.	0x0000000000000000
HvPartitionPropertyEmulatedTimerControl	The periodic timer assist control. For details, see section 15.1.5.	0x0000000000000000
HvPartitionPropertyPmTimerAssist	The PM timer assist definition. For details, see section 15.1.6.	0x0000000000000000
HvPartitionPropertyDebugChannelId	The debug session channel identifier. For details, see section 20.1.1.	0x0000000000000000
HvPartitionPropertyVirtualTlbPageCount	The number of pool pages reserved for the virtual TLB. For details, see section 5.2.6.	
HvPartitionPropertyProcessorVendor	The processor vendor. For details, see section 5.2.7. Read-only.	The vendor of the platform's processors.
HvPartitionPropertyProcessorFeatures	The bitmask representing supported processor features. For details, see section 5.2.8. Default value is the value for the root partition. Read-only for the root partition.	The guest visible features supported and enabled by the platform.
HvPartitionPropertyProcessorXsaveFeatures	The bitmask representing supported processor XSAVE related features. For details, see section 5.2.9. Default value is the value for the root partition. Read-only for the root partition.	The guest visible XSAVE features supported and enabled by the platform.
HvPartitionPropertyProcessorCLFlushSize	The size of the processor cache line flush size. For details, see section 5.2.105.2.10. Default value is the value for the root partition. Read-only for the root partition.	The minimum cache line flush size of all processors on the platform.

5.2.3 Partition Privilege Flags

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 child partition can access.

The property is defined with the following structure:

```
typedef struct
{
// Access to virtual MSRs
UINT64      AccessVpRunTimeMsr:1;
UINT64      AccessPartitionReferenceCounter:1;
```

```
UINT64      AccessSynicMsrs:1;
UINT64      AccessSyntheticTimerMsrs:1;
UINT64      AccessApicMsrs:1;
UINT64      AccessHypercallMsrs:1;
UINT64      AccessVpIndex:1;
UINT64      AccessResetMsr:1;
UINT64      AccessStatsMsr:1;
UINT64      AccessPartitionReferenceTsc:1;
UINT64      AccessGuestIdleMsr:1;
UINT64      Reserved1:`21;

// Access to hypercalls
UINT64      CreatePartitions:1;
UINT64      AccessPartitionId:1;
UINT64      AccessMemoryPool:1;
UINT64      AdjustMessageBuffers: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      ConfigureProfiler:1;
UINT64      Reserved3:18;

} HV_PARTITION_PRIVILEGE_MASK;
```

The following table explains what each of these flags controls. The table also lists the default value for non-root partitions. The root partition has access to all capabilities.

Privilege Flag	Meaning	Default Value
AccessVpRunTimeMsr	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.	1
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.	1
AccessSynicMsrs	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.	1
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.	1
AccessApicMsrs	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.	1
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.	1
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.	1
AccessResetMsr	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.	1

Privilege Flag	Meaning	Default Value
AccessStatsMsr	This partition has access to the synthetic MSRs that allows the guest to map and unmap its own statistics pages.	1
AccessPartitionReferenceTsc	The partition has access to the reference TSC.	0
AccessGuestIdleMsr	The partition has access to the synthetic MSR that allows the guest to enter the guest idle state.	1
CreatePartitions	The partition can invoke the hypercall HvCreatePartition. The partition also can make any other hypercall that is restricted to operating on children.	0
AccessPartitionId	The partition can invoke the hypercall HvGetPartitionId to obtain its own partition ID.	0
AccessMemoryPool	The partition can invoke the hypercalls HvDepositMemory, HvWithdrawMemory and HvGetMemoryBalance.	0
PostMessages	The partition can invoke the hypercall HvPostMessage.	0
SignalEvents	The partition can invoke the hypercall HvSignalEvent.	0
CreatePort	The partition can invoke the hypercall HvCreatePort.	0
ConnectPort	The partition can invoke the hypercall HvConnectPort.	0
AccessStats	The partition can invoke the hypercalls HvMapStatsPage and HvUnmapStatsPage.	0
Debugging1	The partition can invoke the hypercalls HvPostDebugData, HvRetrieveDebugData and HvResetDebugSession.	0
CpuManagement [1]	The partition can invoke the hypercalls HvGetLogicalProcessorRunTime andHvCallParkLogicalProcessors. This partition also has access to the power management MSRs.	0

5.2.4 Partition Creation Flags

When a partition is created, the parent has the option of specifying a *creation flags* mask. Creation flags are reserved for future use and should be set to zero.

5.2.5 Partition State

Each partition is in one of four states:

Uninitialized – The state of the partition after it has been created by a call to HvCreatePartition. Only the HvDepositMemory, HvQueryMemoryBalance, HvGetNextChildPartition and HvInitializePartition hypercalls are permitted on *uninitialized* partitions.

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

Active – The state of the partition after it has been successfully initialized by a call to HvInitializePartition. All hypercalls are permitted on *active* partitions, with the exception of HvDeletePartition.

Finalizing – The transitory state of the partition after HvFinalizePartition has been invoked but not yet completed. Only the HvQueryMemoryBalance, HvWithdrawMemory, HvGetNextChildPartition and HvFinalizePartition hypercalls are permitted on *finalizing* partitions.

Finalized – The state of the partition after it has been finalized by a successfully completed call to HvFinalizePartition. Only the HvQueryMemoryBalance, HvWithdrawMemory, HvGetNextChildPartition and HvDeletePartition hypercalls are permitted on *finalized* partitions.

Figure 1 is a state diagram that shows the valid states and actions that can result in state transitions. All state transitions appear to be atomic from the perspective of hypervisor callers.

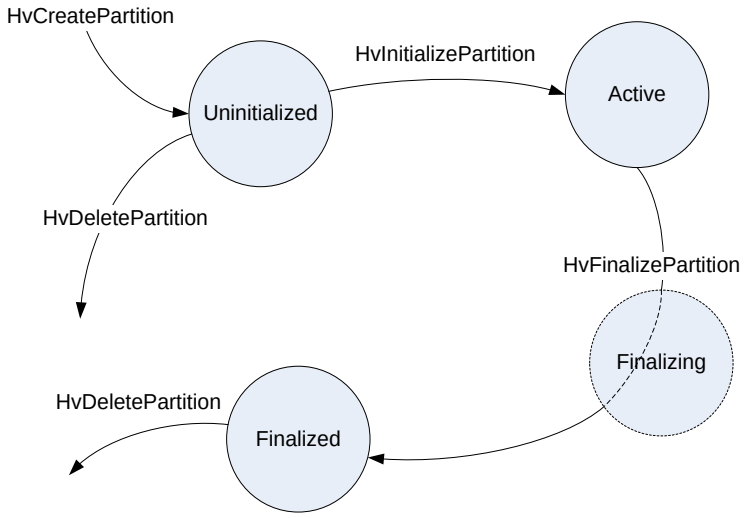


Figure 1. Partition management state diagram

5.2.6 Partition Virtual TLB Page Count

The hypervisor maintains a virtual TLB for each virtual processor (see section 12.1). The memory available for the virtual TLB is reserved and allocated from the partition's memory pool. The count of pages reserved is a partition property that can be modified dynamically. The hypervisor establishes a default value, which may be different for the root and other (non-root) partitions. It also enforces minimum and maximum page count restrictions. Attempts to set the page count outside of the restrictions will result in the value being set to the appropriate minimum or maximum value.

5.2.7 Partition Processor Vendor

One of the partition properties (HvPartitionProcessorVendor) defines the processor vendor of the platform the partition was created on.

The codes are defined as follows.

```
typedef enum _HV_PROCESSOR_VENDOR
```

```
    HvProcessorVendorAmd            = 0x0000,
    HvProcessorVendorIntel          = 0x0001

} HV_PROCESSOR_VENDOR, *PHV_PROCESSOR_VENDOR;
```

5.2.8 Partition Processor Features

One of the partition properties (HvPartitionProcessorFeatures) defines the processor features that are verified for migration compatibility when restoring a VM[2]. This allows a partition's processor features to be enumerated or de-featured, to enable platform migration.

The property is defined with the following structure:

```
typedef _HV_PARTITION_PROCESSOR_FEATURES
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Sse3Support:1;
        UINT64 LahfSahfSupport:1;
        UINT64 Ssse3Support:1;
        UINT64 Sse4_1Support:1;
        UINT64 Sse4_2Support:1;
        UINT64 Sse4aSupport:1;
        UINT64 Sse5Support:1;
        UINT64 PopCntSupport:1;
        UINT64 Cmpxchg16bSupport:1;
        UINT64 Altmovcr8Support:1;
        UINT64 LzcntSupport:1;
        UINT64 MisAlignSseSupport:1;
        UINT64 MmxExtSupport:1;
        UINT64 Amd3DNowSupport:1;
        UINT64 ExtendedAmd3DNowSupport:1;
        UINT64 Reserved1:48;
    };

} HV_PARTITION_PROCESSOR_FEATURES,
*PHV_PARTITION_PROCESSOR_FEATURES;
```

5.2.9 Partition Processor XSAVE Features

Similar to HvPartitionProcessorFeatures, HvPartitionProcessorXsaveFeatures defines the processor XSAVE related features that are verified for migration compatibility when restoring a VM processor. This allows a partition's processor XSAVE features to be enumerated or defeatured, to enable VM migration.

The property is defined with the following structure:

```
typedef union _HV_PARTITION_PROCESSOR_XSAVE_FEATURES
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 XsaveSupport:1;
```

[2] This feature set is a subset of all processor features as exposed by CPUID. Other features are either hidden from non-root guests or are required to be enabled for Hyper-V to launch.

```
        UINT64 XsaveoptSupport:1;
        UINT64 AvxSupport:1;
        UINT64 Reserved1:61;
    };
HV_PARTITION_PROCESSOR_XSAVE_FEATURES,
*PHV_PARTITION_PROCESSOR_XSAVE_FEATURES;
```

5.2.10 Partition Cache Line Flush Size

This property defines the cache line flush size of the partition. This allows a partition's processor cache line flush size to be enumerated or specified, to enable VM migration.

This property is a UINT64,

5.2.11 Partition Compatibility Mode

To allow a VM to migrate between platforms of the same processor architecture but with differing processor feature sets and characteristics, the following defines the partition properties for compatibility VM.

```
#define HV_PARTITION_PROCESSOR_FEATURES_INTEL_COMPATIBILITY_MODE \
{   1,   /* Sse3Support */ \
    1,   /* LahfSahfSupport */ \
    0,   /* Ssse3Support */ \
    0,   /* Sse4_1Support */ \
    0,   /* Sse4_2Support */ \
    0,   /* Sse4aSupport */ \
    0,   /* Sse5Support */ \
    0,   /* PopCntSupport */ \
    1,   /* Cmpxchg16bSupport */ \
    0,   /* Altmovcr8Support */ \
    0,   /* LzcntSupport */ \
    0,   /* MisAlignSseSupport */ \
    0,   /* MmxExtSupport */ \
    0,   /* Amd3DNowSupport */ \
    0,   /* ExtendedAmd3DNowSupport */ \
    0    /* Reserved1 */ \
}

#define HV_PARTITION_PROCESSOR_FEATURES_AMD_COMPATIBILITY_MODE \
{ \
    1,   /* Sse3Support */ \
    1,   /* LahfSahfSupport */ \
    0,   /* Ssse3Support */ \
    0,   /* Sse4_1Support */ \
    0,   /* Sse4_2Support */ \
    0,   /* Sse4aSupport */ \
    0,   /* Sse5Support */ \
    0,   /* PopCntSupport */ \
    1,   /* Cmpxchg16bSupport */ \
    1,   /* Altmovcr8Support */ \
    0,   /* LzcntSupport */ \
    0,   /* MisAlignSseSupport */ \
    1,   /* MmxExtSupport */ \
    0,   /* Amd3DNowSupport */ \
    0,   /* ExtendedAmd3DNowSupport */ \
    0    /* Reserved1 */ \
}

typedef union _HV_PARTITION_PROCESSOR_XSAVE_FEATURES
{
    struct
    {
```

```
        UINT64 XsaveSupport:1;
        UINT64 XsaveoptSupport:1;
        UINT64 AvxSupport:1;
        UINT64 Reserved1:60;
    };
    UINT64 AsUINT64;

} HV_PARTITION_PROCESSOR_XSAVE_FEATURES,
*PHV_PARTITION_PROCESSOR_XSAVE_FEATURES;

#define HV_PARTITION_PROCESSOR_XSAVE_FEATURES_INTEL_COMPATIBILITY_MODE
\
{ \
    0,   /* XsaveSupport */ \
    0,   /* XsaveoptSupport */ \
    0,   /* AvxSupport */ \
    0    /* Reserved1 */ \
}

#define HV_PARTITION_PROCESSOR_XSAVE_FEATURES_AMD_COMPATIBILITY_MODE \
{ \
    0,   /* XsaveSupport */ \
    0,   /* XsaveoptSupport */ \
    0,   /* AvxSupport */ \
    0    /* Reserved1 */ \
}

#define HV_PARTITION_PROCESSOR_CL_FLUSHSIZE_INTEL_COMPATIBILITY_MODE
(8)
#define HV_PARTITION_PROCESSOR_CL_FLUSHSIZE_AMD_COMPATIBILITY_MODE (8)
```

5.3 Partition Creation

Partitions creation involves several steps:

HvCreatePartition allocates a new child partition using the minimal amount of memory from the parent's memory pool. This memory is used to create the basis for a partition that can subsequently be provisioned and initialized. As part of creation, the new partition is assigned a partition ID. This ID is used to identify the partition during all subsequent steps.
Memory is added to the newly-created partition's memory pool by calling HvDepositMemory.
HvInitializePartition initializes the partition. After successfully initialized, the partition enters the "active" state and additional operations can be performed on the partition.
Additional resources are added to the partition. For example, one or more virtual processors can be allocated, memory can be mapped to the partition's GPA space, and scheduling policies can be defined.

5.4 Partition Destruction

Partition destruction is complicated by the fact that memory must be reclaimed before the partition disappears. Furthermore, a parent partition might not have the rights to access some of the memory while its child is running, such as the memory used for various internal hypervisor data structures. Destruction is thus a two-part operation, finalization followed by deletion.

5.4.1 Partition Finalization

Finalization of a partition causes an irreversible change in the partition state. A parent can request this state change for any of its children by calling HvFinalizePartition.

Partition finalization takes an indeterminate amount of time depending on the amount of internal cleanup that is required to bring the partition to the *finalized* state. Cleanup de-allocates the partition's internal hypervisor data structures — including those structures associated with virtual

processors, intercepts, ports, connections, GPA mappings, and memory buffers — and returns them to the pool from which they were allocated. Once the finalization process has begun, very few operations can be performed on the partition.

A finalized partition remains in that state until it is *deleted*. A partition in the finalized state has the following characteristics:

- All virtual processors belonging to the partition are completely deleted. A finalized partition is unable to execute any guest instructions.
- The partition does not receive messages or interrupts. Attempts to send messages to the partition will fail.
- Any hypervisor call that has the partition identifier as one of its parameters will fail, except for HvQueryMemoryBalance, HvWithdrawMemory, HvDeletePartition and HvGetNextChildPartition.

5.4.2 Partition Deletion

After a partition is finalized, the parent must reclaim the resources associated with the target partition. In general, the parent might not be authorized to perform these actions *before* the partition is finalized. Note that resources associated with descendants of the target partition have been implicitly withdrawn during finalization.

Resources are reclaimed by calling HvWithdrawMemory, which should be called as many times as required to reclaim all resources. After the resources have been reclaimed, the target can be deleted by calling HvDeletePartition.

5.4.3 Partition Destruction

The full algorithm for destroying a partition is as follows:

- Initiate the finalization process by calling HvFinalizePartition for the target partition. This brings the target partition to the finalized state.
- Reclaim all resources by calling HvWithdrawMemory for the target partition as many times as necessary.
- Delete the partition by calling HvDeletePartition for the target partition.

5.5 Partition Enumeration

The hypervisor provides a hypercall interface to enumerate the partition IDs of children of a particular parent partition. In order for the enumeration process to produce predictable results, the hypervisor will return the partition IDs of each child for a specified parent partition in the order of its creation (that is, the more recently that a partition was created, the later it will be returned by an enumeration process).

5.6 Partition Management Interfaces

5.6.1 HvCreatePartition

The HvCreatePartition hypercall allows an authorized guest to create a new partition.

Wrapper Interface

```
HV_STATUS
HvCreatePartition(
      __in  UINT64      Flags,
      __in  HV_PROXIMITY_DOMAIN_INFO            ProximityDomainInfo,
      __out PHV_PARTITION_ID                    NewPartitionId
);
```

Native Interface

HvCreatePartition	
	Call Code = 0x0040
➡ Input Parameters	
0	Flags (8 bytes)
8	ProximityDomainInfo (8 bytes)
⬅ Output Parameters	
0	NewPartitionId (8 bytes)

Description

Partition creation requires the hypervisor to allocate a new partition data structure. The memory required for the partition's initial data structure comes from the caller's memory pool. For information on memory pools, see section 7.1.1.

If a partition is successfully created, it begins in the "uninitialized" state. The new partition's memory pool is initially empty and the caller must populate it with sufficient pages to allow a subsequent HvInitializePartition hypercall to succeed.

The hypervisor restricts the number maximum depth of the partition hierarchy. All hypervisor implementations must support a depth of at least two partitions (that is, the root partition and one descendent).

Input Parameters

Flags is a mask of capabilities that are permanently assigned to the new partition. Creation flags are described in section 5.2.4. They are reserved for future use and should be set to zero.

ProximityDomainInfo specifies the ACPI proximity domain information of the NUMA node where the partition's initial hypervisor data structures will reside. If there are no pages in the caller's pool for the specified ID, then the call will fail. The Proximity Domain specification is described in section 7.2.1.

Output Parameters

NewPartitionId is the identifier for the new partition. This value is guaranteed to be unique within the current power cycle.

Restrictions

The caller must possess the *CreatePartitions* privilege.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller's partition privilege flag CreatePartitions is cleared.
HV_STATUS_INVALID_PARAMETER	The *flags* parameter includes bits for capabilities that are either undefined or reserved in the implementation..
	The *ProximityDomainInfo* parameter specifies an invalid flag bit or an invalid domain ID.
HV_STATUS_PARTITION_TOO_DEEP	Creating a new partition would exceed the maximum depth in the partition hierarchy.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the caller is insufficient to perform the operation.
HV_STATUS_NO_RESOURCES	A required system resource is unavailable or an implementation limit has been reached.

5.6.2 HvInitializePartition

The HvInitializePartition hypercall allows a parent to initialize a child partition.

Wrapper Interface

```
HV_STATUS
HvInitializePartition(
      __in HV_PARTITION_ID    PartitionId
);
```

Native Interface

HvInitializePartition [fast]	
	Call Code = 0x0041
➡ Input Parameters	
0	PartitionId (8 bytes)

Description

The HvInitializePartition call is used to bring a partition from the "uninitialized" to the "active" state. The child partition must be in the "uninitialized" state; that is, it must have been allocated with a call to HvCreatePartition. Prior to using the HvInitializePartition hypercall, the caller may be required to initially populate the partition's memory pool (see section 7.1.1). The internal data structures allocated by this hypercall are allocated preferentially from the proximity domain specified by the caller in the related HvCreatePartition hypercall.

Input Parameters

PartitionId is the partition ID of the child partition that is to be initialized.

Output Parameters

None.

Restrictions

The partition specified by *PartitionId* must be in the "uninitialized" state.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The specified partition is not a child of the caller.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "uninitialized" state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the specified partition is insufficient to perform the operation.
HV_STATUS_NO_RESOURCES	A required system resource is unavailable or an implementation limit has been reached.

5.6.3 HvFinalizePartition

The HvFinalizePartition hypercall begins the partition destruction operation by freeing all internal hypervisor resources.

Wrapper Interface

```
HV_STATUS
HvFinalizePartition(
      __in HV_PARTITION_ID    PartitionId
);
```

Native Interface

HvFinalizePartition [fast]	
	Call Code = 0x0042
➡ Input Parameters	
0	PartitionId (8 bytes)

Description

This call begins the irreversible destruction process of a partition and its children. It frees internal hypervisor data structures, returning the memory to the partition's memory pool so the pages can be subsequently withdrawn.

Finalization processing for a partition and its descendants takes an indeterminate amount of time. The architectural state of the partition(s) involved will be in transition until the process completes.

The caller must be the parent of the specified partition. In addition, the partition must be in either the "active" or "finalizing" state. For more information on partition destruction, see section 5.4.

Input Parameters

PartitionId specifies the partition to be finalized.

Output Parameters

None.

Restrictions

The specified partition must be in the "active" or "finalizing" state.

The caller must be the parent of the specified partition.

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	*PartitionId* is invalid.
HV_STATUS_OPERATION_DENIED	The specified partition has one or more children.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

5.6.4 HvDeletePartition

The HvDeletePartition hypercall completes the partition destruction operation by deleting a finalized or uninitialized partition.

Wrapper Interface

```
HV_STATUS
HvDeletePartition(
      __in HV_PARTITION_ID    PartitionId
);
```

Native Interface

HvDeletePartition [fast]	
	Call Code = 0x0043
➡ Input Parameters	
0	PartitionId (8 bytes)

Description

The caller must be the parent of the specified partition. Partition deletion releases any memory allocated by HvCreatePartition back to the creating partition's pool. The deleted partition's resources are returned as described in section 5.4.

After a successful deletion, the partition ID is considered invalid.

Input Parameters
PartitionId specifies the partition to be deleted.

Output Parameters
None.

Restrictions
The caller must be the parent of the partition specified by *PartitionId.*
The partition specified by *PartitionId* must be in the "finalized" or "uninitialized" state.
The partition's pool must be empty.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition.
HV_STATUS_OPERATION_DENIED	The specified partition's pool has not been drained of all pages.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "finalized" or "uninitialized" state.
HV_STATUS_INVALID_PARTITION_ID	*PartitionId* is invalid.

5.6.5 HvGetPartitionProperty

The HvGetPartitionProperty hypercall allows an authorized guest to read a property of a partition.

Wrapper Interface

```
HV_STATUS
HvGetPartitionProperty(
      __in  HV_PARTITION_ID   PartitionId,
      __in  HV_PARTITION_PROPERTY_CODE    PropertyCode,
      __out PHV_PARTITION_PROPERTY  PropertyValue
);
```

Native Interface

HvGetPartitionProperty		
	Call Code = 0x0044	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	PropertyCode (4 bytes)	Padding (4 bytes)
⬅ Output Parameters		
0	PropertyValue (8 bytes)	

Description

The caller must be the specified partition's parent, or the root partition requesting its own properties.

Input Parameters

PartitionId specifies a partition.

PropertyCode specifies the property the caller is interested in retrieving.

Output Parameters

PropertyValue returns the property value.

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 root partition specifying its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified target partition (unless the caller and the target partition are both the root partition).
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_UNKNOWN_PROPERTY	The specified property code is not a recognized property.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.

5.6.6 HvSetPartitionProperty

The HvSetPartitionProperty hypercall allows a parent partition to modify a property of one of its children.

Wrapper Interface

```
HV_STATUS
HvSetPartitionProperty(
      __in HV_PARTITION_ID    PartitionId,
      __in HV_PARTITION_PROPERTY_CODE     PropertyCode,
      __in HV_PARTITION_PROPERTY    PropertyValue
);
```

Native Interface

HvSetPartitionProperty		
	Call Code = 0x0045	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	PropertyCode (4 bytes)	Padding (4 bytes)
16	PropertyValue (8 bytes)	

Description

The caller must be the specified partition’s parent, or the root partition requesting its own properties.

Input Parameters

PartitionId specifies a partition.

PropertyCode specifies the property the caller is interested in modifying.

PropertyValue specifies the new property value.

Output Parameters

None.

Restrictions

The specified partition must be in the “active” state.

The caller must either be the parent of the partition specified by *PartitionId* or the root partition specifying its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified target partition (unless the caller and the target partition are both the root partition).
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_UNKNOWN_PROPERTY	The specified property ID is not a recognized property.
HV_STATUS_PROPERTY_VALUE_OUT_OF_RANGE	The specified property value, while valid in some circumstances, violates an invariant (for example, a CPU reservation causes the total reservations to exceed 100%).
	An attempt was made to modify a read-only partition privilege.
	An attempt was made to set the *CpuManagement* privilege on a hypervisor implementation that reserves this privilege to the root partition.
HV_STATUS_OBJECT_IN_USE	The specified property value, when non-zero, is required to be unique across all partitions but is currently in use.
HV_STATUS_INVALID_PARAMETER	The specified property value is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the specified partition is insufficient to perform the operation.

5.6.7 HvGetPartitionId

The HvGetPartitionId hypercall returns the partition ID of the caller.

Wrapper Interface

```
HV_STATUS
HvGetPartitionId(
      __out PHV_PARTITION_ID  PartitionId
);
```

Native Interface

HvGetPartitionId	
	Call Code = 0x0046
⬅ Output Parameters	
0	PartitionId (8 bytes)

Description

The call returns the partition ID of the caller. Note that the partition ID can change if the partition is saved and subsequently restored or if the partition is migrated across systems. In such cases, the *AccessSelfPartitionId* privilege can be used to prevent a guest from obtaining its partition ID.

Input Parameters
None.

Output Parameters
PartitionId provides the partition ID of the caller.

Restrictions
The caller must possess the *AccessPartitionId* privilege.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller's partition privilege flag *AccessSelfPartitionId* is cleared.

5.6.8 HvGetNextChildPartition

The HvGetNextChildPartition hypercall allows callers to enumerate the child partitions for a given parent partition.

Wrapper Interface

```
HV_STATUS
HvGetNextChildPartition(
      __in  HV_PARTITION_ID   ParentId,
      __in  HV_PARTITION_ID   PreviousChildId,
      __out PHV_PARTITION_ID  NextChildId
);
```

Native Interface

HvGetNextChildPartition	
	Call Code = 0x0047
➡ Input Parameters	
0	ParentId (8 bytes)
8	PreviousChildId (8 bytes)
⬅ Output Parameters	
0	NextChildId (8 bytes)

Description

The hypercall returns the partition ID of the next child partition created after the one specified. To enumerate all children of a particular parent partition, the caller should specify HV_PARTITION_ID_INVALID as the value for *PreviousChildId*. Subsequent calls should specify the ID of the previously-returned child ID. When the caller has enumerated all partition IDs, the value HV_PARTITION_ID_INVALID will be returned in *NextChildId*. If a child partition is deleted during enumeration, the results are dependent on whether the caller has already enumerated past the deleted partition. Callers should be prepared to handle an HV_STATUS_INVALID_PARTITION_ID error condition which might indicate that the partition identified by *PreviousChildId* has been deleted.

Partitions are enumerated by the hypervisor in the order of their creation.

Input Parameters
ParentId specifies the ID of the parent partition whose children are being enumerated.

PreviousChildId specifies the ID of the previous child partition to be enumerated or HV_PARTITION_ID_INVALID to specify that the first child partition should be returned.

Output Parameters

NextChildId indicates the next child partition in the list or HV_PARTITION_ID_INVALID if the previous partition was the final child partition.

Restrictions

The partition specified by *ParentId* must be the parent of the partition specified by *PreviousChildId.*.

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 parent ID is invalid.
	The specified *PreviousChildId* is not the value HV_PARTITION_ID_INVALID and is not a valid child partition of the partition identified by *ParentId*.

6 Physical Hardware Management

6.1 Overview

This section discusses the way in which the hypervisor manages aspects of the underlying physical hardware.

The hypervisor does not manage or control all hardware in the system. Rather, it manages a small subset of core hardware facilities—just enough to enforce isolation guarantees. In some cases, the hypervisor delegates hardware management responsibilities to the root partition.

The following table shows which aspects of hardware management are handled by the hypervisor, by the root partition, and (optionally) by other partitions.

Software Component	Hardware Facility
Hypervisor	Logical processors (scheduling). Local APICs (interrupt routing). Constant-rate system counter (for example, PM timer). System physical address space (access to RAM and device memory). I/O space. MSR space.
Root partition	Processor power management. System power management. Memory and processor hot add and removal. Device hot add and removal. PCI, PCIe and PCIx configuration space. Devices connected directly to a PCIe bus or to a PCIe-attached peripheral bus.

6.1.1 System Physical Address Space

The *system physical address space* (or SPA space) of the physical machine is the range from 0 to some maximum address that depends on the underlying hardware architecture—typically 2^n-1, where $32 \le n \le 64$.

SPAs may be backed by RAM or memory-mapped device registers at the granularity of the architecturally-defined minimum page size HV_PAGE_SIZE. The layout of the SPA space is ultimately dictated by the physical machine.

SPA pages are specified in terms of *SPA page ranges,* each of which consists of one or more contiguous SPA pages. RAM SPA ranges are backed by normal memory.

It is a critical invariant for the correct functioning of the hypervisor that all RAM SPA pages are backed by RAM at all times; that is, read and write accesses to these pages should have the expected semantics. The hypervisor relies on the correct behavior of a privileged partition (typically the root partition) in ensuring that this invariant holds. Future versions of the hypervisor may also rely on hardware features to help enforce this invariant.

The following invariants are assumed (and enforced) by the hypervisor:

- No two RAM page ranges can overlap.
- Pages that are not within defined RAM page ranges are assumed to be “not backed” or backed by device memory and are not used by the hypervisor. They are inaccessible to all partitions except for the root partition.
- The hypervisor uses only pages within RAM SPA page ranges to store its code and internal data structures.

6.1.2 Logical Processors

The hypervisor maintains a list of *logical processors.* Conceptually, a logical processor is a thread implemented in hardware.

At hypervisor boot time, one virtual processor is created within the root partition for each present and potential logical processor. Unlike normal virtual processors associated with other partitions, the root partition's virtual processors have hard affinities; that is, they are bound to a specific logical processor. Partitions possessing the *CpuManagement* privilege (see section 5.2.3) also have hard affinities.

6.1.3 Dynamic Addition of Logical Processors

The hypervisor does not support dynamic addition and removal of logical processors. All potential logical processors must be declared at boot time.

6.1.4 Physical Nodes

A *physical node* represents a collection of logical processors and system physical pages. A physical node is sometimes referred to as a *proximity domain* because all logical processors and pages within the node are assumed to have similar proximity (that is, similar access times to intra-node hardware resources).

The hypervisor maintains a map between pairs of physical nodes that indicate the "distance" (that is, the cost of accessing resources) between the nodes.

6.1.5 System Reset

The hypervisor provides an interface for "enlightened reboot", that is, a mechanism through which a guest may request the hypervisor to reboot the hardware platform. The interface is provided through an MSR that is accessible on all virtual processors of a partition that possesses the *AccessResetMsr* privilege.

6.2 Hardware Information

The hypervisor obtains information about the underlying physical hardware from three sources:

Boot-time input parameters: When the hypervisor is booted, input parameters provide information about some aspects of the physical hardware.

Dynamic discovery: At run-time, the hypervisor can discover information about the underlying physical hardware by using architecturally-defined mechanisms (for example, on x64 platforms, the CPUID instruction).

Root partition input: For more complex dynamic hardware changes (for example, power management and hot add / removal) the hypervisor must be told about hardware changes by code running within the root partition.

6.2.1 Boot-Time Hardware Properties

Aspects of the hardware that are described to the hypervisor at boot time include:

Present and potential logical processors, including those that may be hot-plugged at run-time

Whether hyperthreading is enabled or disabled in the BIOS

Present RAM SPA ranges—system physical address ranges that are populated with RAM when the hypervisor is booted

Physical nodes (including those that have no associated resources at boot time but may be populated at run-time)

Memory access ratios between physical nodes

Addresses of certain hardware features that the hypervisor must access (for example, the power management timer)

For more information about the hypervisor boot process and boot-time input parameters, refer to Chapter 22.

6.2.2 Discovered Hardware Properties

Hardware properties discovered or derived by the hypervisor at run-time include:

The relationship between logical processors (for example, multi-core, hyperthreading)

6.2.3 Root Partition Hardware Properties

The root partition is responsible for notifying the hypervisor of the following hardware properties:

Power management state changes of logical processors

6.3 Hardware Management Data Types

6.3.1 Logical Processors

Logical processors are defined by a 32-bit index.

```
typedef UINT32 HV_LOGICAL_PROCESSOR_INDEX;
```

6.3.2 Power States

Each logical processor has an associated power state, which can be changed by the root partition as its policy dictates. The root partition uses power management configuration MSRs to communicate the recipe for power state transitions for each logical processor, and for informing the hypervisor when a power state change is to be triggered.

The hypervisor supports transitions to and from power states C1, C2, and C3.

6.3.2.1 Power State MSRs

Power state transitions are configured by using model-specific registers (MSRs) associated with each logical processor. Each of the power states has an associated pair of MSRs.

MSR address	Register name	Function
0x400000D1	HV_X64_MSR_POWER_STATE_CONFIG_C1	Configuration register for power state C1
0x400000C1	HV_X64_MSR_POWER_STATE_TRIGGER_C1	Trigger register for power state C1
0x400000D2	HV_X64_MSR_POWER_STATE_CONFIG_C2	Configuration register for power state C2
0x400000C2	HV_X64_MSR_POWER_STATE_TRIGGER_C2	Trigger register for power state C2
0x400000D3	HV_X64_MSR_POWER_STATE_CONFIG_C3	Configuration register for power state C3
0x400000C3	HV_X64_MSR_POWER_STATE_TRIGGER_C3	Trigger register for power state C3

6.3.2.1.1 Power State Configuration Register

Power state configuration registers are read/write registers which are used by the power management partition to specify the “recipe” for transitioning to and from the corresponding power state.

63:60	59	58	57	56	55:52	51:0
RsvdZ	SetARB_DIS	ClearBM_RST	SetBM_RST	CheckBM_STS	ChangeType	TypeSpecific

SetARB_DIS defines the value for the ACPI power management bit ARB_DIS, which is used to enable and disable the system arbiter.

ClearBM_RST clears the ACPI power management bit BM_RLD, which determines if the power state was exited as a result of bus master requests.

SetBM_RST sets the ACPI power management bit BM_RLD, which determines if the power state was exited as a result of bus master requests.

CheckBM_STS determines the value of the ACPI power management bit BM_STS, which determines the power state to enter when considering a transition to or from the C2/C3 power state..

ChangeType defines the power change method to use. Valid options are listed below, and should be specified using the values in HV_X64_POWER_CHANGE_METHOD.:

Issue HLT – execute a HLT instruction

Read IO then Issue HLT – read from an I/O port and then issue a HLT instruction

Read IO – read from an I/O port

Issue MWAIT – issue an MWAIT instruction.

```
typedef enum _HV_X64_POWER_CHANGE_METHOD
{
    HvX64PowerChangeIssueHlt,
    HvX64PowerChangeReadIoThenIssueHlt,
    HvX64PowerChangeReadIo,
    HvX64PowerChangeIssueMwait
} HV_X64_POWER_CHANGE_METHOD, *PHV_X64_POWER_CHANGE_METHOD;
```

TypeSpecific defines the input format specific to the particular type of power change method being requested.

Issue HLT

63:52	51:0
Common input - see above	RsvdZ

Read I/O
Read I/O then issue HLT

63:52	51:16	0:15
Common input - see above	RsvdZ	Port

Port designates the I/O port to read from.

Issue MWAIT

63:52	51:33	32	31:0
Common input - see above	RsvdZ	BreakOnMaskedInterrupt	Hints

BreakOnMaskedInterrupt specifies that receipt of a masked interrupt should cause a break event.

Hints defines the hints to the MWAIT instruction specifying the lower power state to enter.

6.3.2.1.2 Power State Trigger Register

Power state trigger registers are read-only registers that cause the process to enter the corresponding power state.

63:56	55:1	0
IdleEntryCount	RsvdZ	ActiveBM_STS

IdleEntryCount contains the number of times that the logical processor went idle since the power management partition took its affinity bound virtual processor idle.

ActiveBM_STS specifies the value of the ACPI power management bit BM_STS, which determines the power state to enter when considering a transition to or from the C2/C3 power state.

6.3.3 Logical Processor Run Time

Associated with each logical processor is a 64-bit quantity called a *run time*. The run time is defined as the number of 100ns time units that have elapsed since the logical processor was last reset (through the APIC) and during which it was actively executing code (either guest code or hypervisor code). When a logical processor has no code to run or enters a low-power or offline state, its run time counter effectively stops.

The following type is used for the run-time value.

```
typedef UINT64 HV_NANO100_TIME;
typedef HV_NANO100_TIME *PHV_NANO100_TIME;
```

6.3.4 Global Run Time

The hypervisor also maintains a global run time that starts when the hypervisor is booted. As with the logical processor run time, it is recorded in 100ns time units.

6.3.5 System Reset MSR

A partition that possesses the *AccessSystemResetMsr* privilege has access to the system reset MSR, defined as follows:

```
#define HV_X64_MSR_RESET 0x40000003
```

If the MSR is available to the guest, it is accessible from all of the partition's virtual processors. The register has the following format:

63:1	0
RsvdZ	Reset

The MSR always reads as zero. When Reset is written as one, the hypervisor will perform the requested reboot operation.

6.4 Hardware Management Interfaces

6.4.1 HvGetLogicalProcessorRunTime

The HvGetLogicalProcessorRunTime hypercall allows an authorized partition to query the *run time* of the logical processor associated with the caller's virtual processor.

Wrapper Interface

```
HV_STATUS
HvGetLogicalProcessorRunTime(
      __out     PHV_NANO100_TIME    GlobalTime
      __out     PHV_NANO100_TIME    LocalRunTime,
      __out PHV_NANO100_TIME  HypervisorTime
);
```

Native Interface

HvGetLogicalProcessorRunTime	
	Call Code = 0x0004
⬅ Output Parameters	
0	GlobalTime (8 bytes)
8	LocalRunTime (8 bytes)
16	RsvdZ (8 bytes)
24	HypervisorTime (8 bytes)

Description

The HvGetLogicalProcessorRunTime hypercall requires that the calling partition possess the *CpuManagement* privilege. For more information on this flag and its application, see section 5.2.3.

For a definition of *run time*, see section 6.3.3. Utilization can be computed by calling this function twice dividing ($LocalRunTime_n$ – $LocalRunTime_{n-1}$) by ($GlobalTime_n$ – $GlobalTime_{n-1}$).

This call can be used to determine whether the hypervisor is actively scheduling activity on the logical processor or whether it is idle and can be placed into a low-power state without negatively impacting the system's performance.

Input Parameters

None.

Output Parameters

GlobalTime provides the current global reference time.

LocalRunTime provides the run time of the logical processor associated with the caller's virtual processor.

HypervisorTime provides the amount of time the hypervisor has executed on the logical processor.

Restrictions

The partition must possess the *CpuManagement* privilege.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller's partition does not possess the *CpuManagement* privilege.

6.4.2 HvParkLogicalProcessors

The HvParkLogicalProcessors hypercall allows an authorized partition to inform the Hypervisor of the set of processors that may be *parked*.

Wrapper Interface

```
HV_STATUS
HvParkLogicalProcessors(
      _in HV_INPUT_PARK_LOGICAL_PROCESSORS ProcessorMask
);
```

Native Interface

HvParkLogicalProcessors		
	Call Code = 0x0009	
➡ Input Parameters		
0	ProcessorMask (8 bytes)	

Description

The HvParkLogicalProcessors hypercall allows an authorized partition to inform the Hypervisor of the set of processors that may be *parked*. A processor that is *parked* may be placed into a low power processor idle sleep state. This scheduling hint allows the hypervisor to attempt to not schedule threads on parked logical processors, increasing energy efficiency.

Input Parameters

ProcessorMask – Specifies set of processors the root partition has decided to park.

Output Parameters

None.

Restrictions

The caller must possess the CpuManagement privilege.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller does not possess the CpuManagement privilege.

7 Resource Management

7.1 Overview

One of the design goals of the hypervisor is to provide availability guarantees to guests. For example, multiple servers that have been consolidated onto a single physical machine should not be able to prevent each other from making progress. It should be possible to run a partition that provides telephony or DVR support such that this partition continues to function irrespective of the potentially adversarial actions of other partitions. This section addresses only the memory management aspects of this problem and only as far as the hypervisor is involved.

In accordance with the overall microkernel architecture of the hypervisor, support for specific resource policy types, such as quotas or reserves, are moved from the hypervisor into guests. The hypervisor must provide minimal support for resource management in connection with several hypercalls that “consume” memory, where “consume” means that the calls may need to allocate memory for internal data structures and that this memory remains allocated beyond the duration of the call.

The resource management state affects, and is affected by, any hypercall that needs to allocate or free memory in the hypervisor. Whether and how a particular call interacts with the resource management state are described as part of the specification for each call. For example, the section on partition creation will specify from whose memory pool the initial resources needed to set up a partition will be deducted.

7.1.1 Memory Pools

For each partition, the hypervisor maintains a *memory pool* of RAM SPA pages. This pool is managed like a bank account. The number of pages in the pool is referred to as the *balance*. Pages can be *deposited* into or *withdrawn* from the pool.

When a partition makes a hypercall that requires memory, the hypervisor draws the required memory from the pool. If the balance in the pool is insufficient, the call fails. If such a hypercall is made by one guest *on behalf of* another guest (in another partition), the hypervisor draws the required memory from the pool of the latter partition. The phrase “on behalf of” refers to any hypervisor call with a HV_PARTITION_ID parameter. Examples of such calls are HvCreateVp and HvMapGpaPages.

Pages within a partition’s memory pool are managed by the hypervisor. These pages cannot be accessed through any partition’s GPA space. That is, in all partitions’ GPA spaces, they must be inaccessible (mapped such that no read, write or execute access is allowed). In general, the only partition that can deposit into or withdraw from a partition is that partition’s parent.

When a partition is created, the memory pool is initially empty. Memory can be deposited by means of the hypercall HvDepositMemory.

The following invariants are maintained:

- All memory pool pages are RAM SPA pages that were previously mapped into the GPA space of the parent partition.
- All pages in a partition’s memory pool must have been explicitly deposited into that partition’s pool; that is, memory does not migrate between memory pools of different partitions.
- Pages in a partition’s memory pool cannot be accessed by any guest; that is, these pages are not mapped into any partition’s GPA space with read, write, or execute privileges.

7.1.2 NUMA Proximity Domains

The ACPI tables include topology information for all processors and memory present in a system at system boot. They also contain information for memory that can be added while the system is running without requiring a reboot. The information identifies sets of logical processors and physical memory ranges, enabling a tight coupling between logical processors and memory ranges. Each coupling is referred to as a *Proximity Domain.* The latency of memory accesses by logical processors within the same proximity domain is typically shorter than the latency of accesses outside of the proximity domain. The root partition can relay this affinity information to the hypervisor by selecting memory resources or specifying the proximity domain to be used with certain hypercalls.

7.2 Resource Management Data Types

7.2.1 Proximity Domains

The following data types are used with proximity domains:

```
// Proximity domain ID is a 32-bit number
typedef UINT32 HV_PROXIMITY_DOMAIN_ID;

// Identifier used when specifying proximity domain information
typedef struct
{
      HV_PROXIMITY_DOMAIN_ID              Id;
      struct
      {
            UINT32                        ProximityPreferred:1;
            UINT32                        Reserved:30;
            UINT32                        ProximityInfoValid:1;
      } Flags;
} HV_PROXIMITY_DOMAIN_INFO;
```

Flags controls the proximity domain information:

ProximityInfoValid indicates the validity of the proximity domain information. If set, the *Id* value and the other flags are valid. Otherwise, a particular proximity domain is not specified and the hypervisor may use any domain.

ProximityPreferred indicates that the ID specifies the preferred proximity domain instead of a required domain. If the hypervisor is unable to complete the operation using that specific domain, it is free to use other domains.

Id specifies the proximity domain ID. All 32-bit values are supported.

7.3 Resource Management Interfaces

7.3.1 HvDepositMemory

The HvDepositMemory hypercall allows a partition to deposit memory into a child partition’s memory pool.

Wrapper Interface

```
HV_STATUS
HvDepositMemory(
      __in    HV_PARTITION_ID PartitionId,
      __inout PUINT32   PageCount,
      __in_ecount(PageCount)
              PCHV_GPA_PAGE_NUMBER  GpaPages
      );
```

Native Interface

HvDepositMemory [rep]	
	Call Code = 0x0048
➡ Input Parameter Header	
0	PartitionId (8 bytes)
➡ Input List Element	
0	GpaPage (8 bytes)

Description

The specified pages become exclusively accessible by the hypervisor. The pages must be mapped within the caller's GPA space with read, write and execute access. If mapped within the GPA space of any other partition, they must be inaccessible. On success, the pages will no longer be accessible to the caller. In addition, future mapping changes to the pages will be prohibited until they have been successfully withdrawn from the pool (using the HvWithdrawMemory hypercall).

The call will fail if a page has already been deposited into a partition's memory pool. If the pages are being used for another purpose, then an error will be returned to indicate that the pages are currently in use. Such purposes include:

- Pages mapped as event log buffers
- Pages locked down for I/O operations

Input Parameters

PartitionId specifies the partition whose memory pool will be credited.

GpaPage specifies the GPA page that is to be deposited into the specified partition's memory pool.

Output Parameters

None.

Restrictions

The caller must possess the *AccessMemoryPool* privilege.

The partition specified by *PartitionId* must be in the "active" or "uninitialized" state.

The caller must either be the parent of the partition specified by *PartitionId* or the root partition specifying its own partition ID.

The caller must possess read, write and execute access to the GPA specified by *GpaPage.*

No other partition may possess read, write or execute access to the GPA specified by *GpaPage.*

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified partition, and the caller's partition privilege flag *AccessMemoryPool* is enabled. 2. The caller is the root partition, and the specified partition is the root partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition is invalid.
HV_STATUS_OPERATION_DENIED	A specified GPA page is not mapped within the caller's GPA space.
	A specified GPA page is not read, write and execute accessible within the GPA space of the caller.
	A specified GPA page is readable, writable or executable within the GPA space of a partition that is not the caller.
	A specified GPA page has already been deposited into the memory pool of a partition.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" or "uninitialized" state.
HV_STATUS_OBJECT_IN_USE	A specified GPA is being used for another purpose and therefore may not be deposited.

7.3.2 HvWithdrawMemory

The HvWithdrawMemory hypercall attempts to remove one or more pages from the memory pool of the specified child partition.

Wrapper Interface

```
HV_STATUS
HvWithdrawMemory(
      __in    HV_PARTITION_ID PartitionId,
      __in    HV_PROXIMITY_DOMAIN_INFO    ProximityDomainInfo,
      __inout PUINT32   PageCount,
      __out_ecount(*PageCount)
              PHV_GPA_PAGE_NUMBER   GpaPages
      );
```

Native Interface

HvWithdrawMemory [rep]	
	Call Code = 0x0049
➡ Input Parameters	
0	PartitionId (8 bytes)
8	ProximityDomainInfo (8 bytes)
⬅ Output List Element	
0	GpaPage (8 bytes)

Description

A page can be withdrawn only if it is not currently in use by the hypervisor. The hypervisor guarantees that the contents of the withdrawn page are zeroed. The GPA for the withdrawn page is the same as the GPA of a page that was previously deposited. On success, the page will be read, write and execute accessible.

If a partition is in the “uninitialized” or “finalized” state, this call is guaranteed to allow withdrawal of all pages within the partition’s pool. In other cases, success is not guaranteed – even if a recent call to HvGetMemoryBalance has indicated the presence of available pages – because the hypervisor may choose to use deposited pages at any time.

Input Parameters

PartitionId specifies the partition whose memory pool will be debited.

ProximityDomainInfo specifies the ACPI proximity domain information of the NUMA node from which the pages are to be withdrawn. The Proximity Domain specifier is described in section 7.2.1.

Output Parameters

GpaPage returns a page that is has been withdrawn from the memory pool.

Restrictions

The caller must possess the *AccessMemoryPool* privilege.

The caller must be the parent of the partition specified by *PartitionId* or the root partition specifying its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified partition, and the caller’s partition privilege flag *AccessMemoryPool* is enabled. 2. The caller is the root partition, and the specified partition is the root partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition is invalid.
HV_STATUS_INVALID_PARAMETER	The *ProximityDomainInfo* parameter specifies an invalid flag bit or an invalid domain ID.
HV_STATUS_NO_RESOURCES	No unused page was available.

7.3.3 HvGetMemoryBalance

The HvGetMemoryBalance hypercall allows a parent partition to query the state of the memory pool of a child partition.

Wrapper Interface

```
HV_STATUS
HvGetMemoryBalance(
      __in  HV_PARTITION_ID               PartitionId,
      __in  HV_PROXIMITY_DOMAIN_INFO      ProximityDomainInfo,
      __out PUINT64                 PagesAvailable,
      __out PUINT64                 PagesInUse
      );
```

Native Interface

HvGetMemoryBalance	
	Call Code = 0x004A
➡ Input Parameters	
0	PartitionId (8 bytes)
8	ProximityDomainInfo (8 bytes)
⬅ Output Parameters	
0	PagesAvailable (8 bytes)
8	PagesInUse (8 bytes)

Description

The caller must either be the root partition or the parent of the partition specified as *PartitionId.*

Input Parameters

PartitionId specifies the partition.

ProximityDomainInfo specifies the ACPI proximity domain information of the NUMA node whose memory balance is being queried. The Proximity Domain specifier is described in section 7.2.1.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified partition, and the caller's partition privilege flag AccessMemoryPool is enabled. 2. The caller is the root partition, and the specified partition is the root partition.
HV_STATUS_INVALID_PARTITION_ID	The specified partition is invalid.
HV_STATUS_INVALID_PARAMETER	The *ProximityDomainInfo* parameter specifies an invalid flag bit or an invalid domain ID.

8 Guest Physical Address Spaces

8.1 Overview

8.1.1 GPA Space

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.

The way in which the GPA space is defined and the behavior associated with unmapped page accesses differs between the root partition and non-root partitions.

For the root partition:

Its GPA space is identity-mapped—that is, all mapped GPA pages map directly to the SPA page with the same address.

The GPA mappings are defined by the hypervisor at boot time or when SPA RAM ranges are added or removed. All valid SPA RAM pages are therefore always mapped within the root partition's GPA space. The root partition is not allowed to unmap these. The root partition may, however, modify its access rights to its mapped pages.

Unmapped GPA pages within the root partition's GPA space (that is, SPA pages that are not part of known SPA RAM ranges) can be accessed by the root partition. This allows the partition to access memory-mapped registers, video memory buffers and so on. Some unmapped pages are off limits even to the root partition. These include SPA pages that correspond to hardware resources that must be managed by the hypervisor for correct operation. For example, the hypervisor does not allow the root partition to directly access the local APIC's memory-mapped registers.

For non-root 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 (through a call to HvMapGpaPages), 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. For a detailed description of the memory access messages, see section 16.2.6.

8.1.2 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 child'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
•		•	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. For a detailed description of the memory access messages, see section 16.2.6.

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.

8.1.3 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.

Parent partitions that include instruction completion logic should use the hypercalls HvTranslateVirtualAddress, HvReadGpa, and HvWriteGpa to emulate virtual processor memory accesses correctly in the presence of overlays.

HvTranslateVirtualAddress returns a flag indicating whether the specified virtual address maps to an overlay page. HvReadGpa and HvWriteGpa perform the GPA access in the same way the specified virtual processor would have. If an overlay page is present at the specified address, the

access is directed to that overlay page. Otherwise, the access is directed to the underlying GPA page.

When the hypervisor performs a guest page table walk either in response to a virtual processor memory access or a call to HvTranslateVirtualAddress, 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.

8.2 GPA Data Types

8.2.1 Map Page Flags

The following flags are used with the HvMapGpaPages hypercall.

```
typedef UINT32 HV_MAP_GPA_FLAGS;

#define HV_MAP_GPA_READABLE   0x00000001
#define HV_MAP_GPA_WRITABLE   0x00000002
#define HV_MAP_GPA_EXECUTABLE 0x00000004
```

8.3 GPA Interfaces

8.3.1 HvMapGpaPages

The HvMapGpaPages hypercall maps one or more GPA pages within the caller's GPA space to pages within the target partition's GPA space.

Wrapper Interface

```
HV_STATUS
HvMapGpaPages(
      __in    HV_PARTITION_ID TargetPartitionId,
      __in    HV_GPA_PAGE_NUMBER    TargetGpaBase,
      __in    HV_MAP_GPA_FLAGS      MapFlags,
      __inout PUINT32   PageCount,
      __in_ecount(PageCount)
              PCHV_GPA_PAGE_NUMBER  SourceGpaPageList
      );
```

Native Interface

HvMapGpaPages [rep]		
	Call Code = 0x004B	
➡ Input Parameter Header		
0	TargetPartitionId (8 bytes)	
8	TargetGpaBase (8 bytes)	
16	MapFlags (4 bytes)	Padding (4 bytes)
➡ Input List Element		
0	SourceGpaPageList (8 bytes)	

Description

HvMapGpaPages is a rep call. The *PageCount* parameter to the C wrapper is passed as the rep count to the hypercall. See section 4.7 for details.

The mapping, if successful, supersedes any previous mapping to the affected GPA pages. Likewise, the specified access rights (readable, writeable and executable) supersede any previously-installed access rights on the GPA pages.

If an existing mapping is being superseded and the pages are being used for another purpose that cannot allow the mapping to be changed, then an error will be returned to indicate that the pages are currently in use. Such purposes include:

Pages mapped as event log buffers
Pages deposited into memory pools
Pages locked down for I/O operations

The GPA pages within the caller's partition must be mapped and not deposited into any partition's memory pool.

A parent partition is allowed to map its GPA pages into a child's GPA space multiple times. It is also allowed to map its GPA pages into the GPA space of multiple children. This allows it to create shared areas between itself and one or more of its children

The root partition is special in that it can call HvMapGpaPages on itself. However, it can modify only the access rights to its GPA pages. It cannot create arbitrary mappings within its own GPA space. Its GPA map is defined by the hypervisor at boot time and at the time SPA RAM ranges are added or removed. Root GPA pages are identity-mapped to SPA pages. Although the root partition can modify the access rights to its own GPA mappings, it cannot establish non-identity mappings.

For best performance on x64 systems, callers should attempt to map at least 2 MB of consecutive GPA pages and make sure that these pages have the same alignment relative to a 2-MB boundary. This will potentially enable the hypervisor to use "large page" mappings and in turn reduce the pressure on physical TLBs. Note that this mapping doesn't need to be done in a single call, and in fact cannot be done in a single call due to the size constraints on the input parameter list.

Input Parameters

TargetPartitionId specifies the target partition whose GPA space is being modified.

TargetGpaBase specifies the base GPA page number in the GPA space of the target partition.

MapFlags specifies the access rights associated with the mapping. Valid access rights flags include HV_MAP_GPA_READABLE, HV_MAP_GPA_WRITABLE, and HV_MAP_GPA_EXECUTABLE. For a description of the valid combinations, see section 8.1.2.

SourceGpaPage specifies a GPA page within the caller's GPA space.

Output Parameters

None.

Restrictions

The partition specified by *TargetPartitionId* must be in the "active" state.

The caller must be the parent of the partition specified by *PartitionId* or the root partition specifying its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified target partition (unless the caller and the target partition are both the root partition).
	The target partition is the root partition and one of the following is true: 1. The target GPA base is not equal to the first source GPA in the list. 2. The source GPA list doesn't consist of contiguous, incrementing pages. 3. The target GPA range references an inaccessible GPA page.
HV_STATUS_INVALID_PARTITION_ID	The specified target partition ID is invalid.
HV_STATUS_INVALID_PARAMETER	A target GPA page lies outside the GPA address space of the target partition.
	A reserved or invalid bit within the specified map flags is set.
	An invalid combination of access rights is specified.
	A source GPA page lies outside the GPA address space of the caller's partition.
HV_STATUS_OPERATION_DENIED	A source GPA page is unmapped.
	A source GPA page has been deposited into a partition's memory pool and is inaccessible.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the partition's memory pool is insufficient to perform the operation.
HV_STATUS_OBJECT_IN_USE	An attempt was made to change the mapping of one or more pages that are being used for a purpose that is incompatible with the change is not permitted.

8.3.2 HvUnmapGpaPages

The HvUnmapGpaPages hypercall unmaps one or more GPA pages from a child partition.

Wrapper Interface

```
HV_STATUS
HvUnmapGpaPages(
      __in    HV_PARTITION_ID TargetPartitionId,
      __in    HV_GPA_PAGE_NUMBER    TargetGpaBase,
      __inout PUINT32   PageCount
      );
```

Native Interface

HvUnmapGpaPages [rep]	
	Call Code = 0x004C
➡ Input Parameter Header	
0	TargetPartitionId (8 bytes)
8	TargetGpaBase (8 bytes)

Description

Note that this call is a “rep” hypercall despite the fact that its parameters do not include an input or output list. The *PageCount* parameter to the C wrapper is passed as the rep count to the hypercall. See section 4.7 for details.

An attempt to unmap an already-unmapped page will be ignored, and a successful status will be returned. Pages deposited to pools are inaccessible and cannot be unmapped until they are first withdrawn (using the HvWithdrawMemory hypercall).

If any of the pages are being used for another purpose that cannot allow the unmapping to be performed, then an error will be returned to indicate that the pages are currently in use. Such purposes include:

- Pages mapped as event log buffers
- Pages deposited into memory pools
- Pages locked down for I/O operations

Unlike HvMapGpaPages, this hypercall cannot be called by the root partition to act upon itself.

Call Type
Rep hypercall

Input Parameters
TargetPartitionId specifies the partition whose GPA space is being modified.

TargetGpaBase specifies the base GPA page number of the GPA range being unmapped.

Output Parameters
None.

Restrictions
The partition specified by *TargetPartitionId* must be in the “active” state.

The caller must be the parent of the partition specified by *TargetPartitionId*.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition.
	The caller is the root partition attempting to unmap pages from its own GPA space.
HV_STATUS_INVALID_PARTITION_ID	The specified target partition ID is invalid.
HV_STATUS_INVALID_PARAMETER	A specified GPA page lies outside the GPA address space of the specified target partition.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_OBJECT_IN_USE	An attempt was made to unmap one or more pages that are being used for a purpose that requires them is not permitted.

8.3.3 HvMapSparseGpaPages

The HvMapSparseGpaPages hypercall maps one or more GPA pages within the caller's GPA space to pages within the target partition's GPA space.

Wrapper Interface

```
typedef struct _HV_GPA_MAPPING
{
    HV_GPA_PAGE_NUMBER TargetGpaPageNumber;
    HV_GPA_PAGE_NUMBER SourceGpaPageNumber;
} HV_GPA_MAPPING, *PHV_GPA_MAPPING;

HV_STATUS
HvMapSparseGpaPages(
      __in    HV_PARTITION_ID TargetPartitionId,
      __in    HV_MAP_GPA_FLAGS      MapFlags,
      __inout PUINT32   PageCount,
      __in_ecount(PageCount)
              PHV_GPA_MAPPING PageList
      );
```

Native Interface

	Call Code = 0x006E	
0	TargetPartitionId (8 bytes)	
8	MapFlags (4 bytes)	Padding (4 bytes)
0	PageList (16 bytes)	

Description

HvMapSparseGpaPages is a rep call. The *PageCount* parameter to the C wrapper is passed as the rep count to the hypercall. See section 4.7 for details.

The mapping, if successful, supersedes any previous mapping to the affected GPA pages. Likewise, the specified access rights (readable, writeable and executable) supersede any previously-installed access rights on the GPA pages.

If an existing mapping is being superseded and the pages are being used for another purpose that cannot allow the mapping to be changed, then an error will be returned to indicate that the pages are currently in use. Such purposes include:

- Pages mapped as event log buffers
- Pages deposited into memory pools
- Pages locked down for I/O operations

The GPA pages within the caller's partition must be mapped and not deposited into any partition's memory pool.

A parent partition is allowed to map its GPA pages into a child's GPA space multiple times. It is also allowed to map its GPA pages into the GPA space of multiple children. This allows it to create shared areas between itself and one or more of its children

The root partition is special in that it can call HvMapSparseGpaPages on itself. However, it can modify only the access rights to its GPA pages. It cannot create arbitrary mappings within its own GPA space. Its GPA map is defined by the hypervisor at boot time and at the time SPA RAM ranges are added or removed. Root GPA pages are identity-mapped to SPA pages. Although the root partition can modify the access rights to its own GPA mappings, it cannot establish non-identity mappings.

For best performance on x64 systems, callers should attempt to map at least 2 MB of consecutive GPA pages and make sure that these pages have the same alignment relative to a 2-MB boundary. This will potentially enable the hypervisor to use "large page" mappings and in turn reduce the pressure on physical TLBs. Note that this mapping doesn't need to be done in a single call, and in fact cannot be done in a single call due to the size constraints on the input parameter list.

Input Parameters

TargetPartitionId specifies the target partition whose GPA space is being modified.

MapFlags specifies the access rights associated with the mapping. Valid access rights flags include HV_MAP_GPA_READABLE, HV_MAP_GPA_WRITABLE, and HV_MAP_GPA_EXECUTABLE. For a description of the valid combinations, see section 8.1.2.

PageList specifies pair of GPA page within the target GPA space and source GPA page within the caller's GPA space.

Output Parameters

None.

Restrictions

The partition specified by *TargetPartitionId* must be in the "active" state.

The caller must be the parent of the partition specified by *PartitionId* or the root partition specifying its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified target partition (unless the caller and the target partition are both the root partition).
	The target partition is the root partition and one of the following is true: 1. The target GPA base is not equal to the first source GPA in the list. 2. The source GPA list doesn't consist of contiguous, incrementing pages. 3. The target GPA range references an inaccessible GPA page.
HV_STATUS_INVALID_PARTITION_ID	The specified target partition ID is invalid.
HV_STATUS_INVALID_PARAMETER	A target GPA page lies outside the GPA address space of the target partition.
	A reserved or invalid bit within the specified map flags is set.
	An invalid combination of access rights is specified.
	A source GPA page lies outside the GPA address space of the caller's partition.
HV_STATUS_OPERATION_DENIED	A source GPA page is unmapped.
	A source GPA page has been deposited into a partition's memory pool and is inaccessible.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the partition's memory pool is insufficient to perform the operation.
HV_STATUS_OBJECT_IN_USE	An attempt was made to change the mapping of one or more pages that are being used for a purpose that is incompatible with the change is not permitted.

9 Intercepts

9.1 Overview

9.1.1 Programmable Intercept Types

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). 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 HvRegisterInterceptSuspend by calling HvSetVpRegisters.

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.

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

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 11.9)	A specific I/O port. The I/O port is specified with each hypercall.	Read *and* Write access flags must be specified to install an intercept.
MSR access (see section 11.10)	All MSRs not being virtualized by the hypervisor. Note that certain privileges affect MSR virtualization. No MSR value is specified with the hypercall.	Read *and* Write access flags must be specified to install the intercept.
CPUID instruction execution (see section 11.11)	A specific CPUID leaf. The CPUID leaf is specified with each hypercall.	Execute access flag must be specified to install an intercept.
Exceptions (see section 11.12)	A specific exception vector. The exception vector is specified with each hypercall.	Execute access flag must be specified to install an intercept.

9.1.2 Unsolicited Intercept Types

Several conditions always cause a virtual processor to be suspended and an intercept message to be sent to the parent. If certain events occur within the root partition, which has no parent, the condition is considered fatal, and the system is restarted.

The following table describes the intercepts defined on x64 implementations of the hypervisor.

Unsolicited Intercept Type	Root Partition Behavior
Invalid virtual processor state (see section 10.1.2)	Fatal error. The system is restarted.
Unrecoverable processor exception (for example, triple fault on X64)	Fatal error. The system is restarted.
Unsupported functionality error	Fatal error. The system is restarted.
FERR asserted (legacy floating point error)	Ignored. No message is generated.
APIC EOIs	Ignored. No message is generated.

Unsupported functionality errors are delivered to the parent if the guest uses a feature of the underlying processor architecture that is not virtualized by the hypervisor and cannot otherwise be reported as “not implemented”. For example, on the x64 architecture, some features can be reported as “not implemented” by using CPUID feature bits or by generating a #GP fault when accessing an MSR. If there is no architectural way for a guest to determine whether a feature is supported, the hypervisor may detect the use of the unsupported feature and deliver an “unsupported functionality” error to the parent.

9.2 Intercept Data Types

9.2.1 Intercept Types

This data type enumerates the intercept classes the hypervisor recognizes.

```
typedef enum
{
      // Platform-specific intercept types
      HvInterceptTypeX64IoPort      = 0x00000000,
      HvInterceptTypeX64Msr   = 0x00000001,
      HvInterceptTypeX64Cpuid = 0x00000002,
      HvInterceptTypeX64Exception   = 0x00000003

} HV_INTERCEPT_TYPE;
```

9.2.2 Intercept Parameters

The following data structures are used to describe information about installed intercepts. Reserved fields must be set to zero.

```
// Used with intercepts of type HV_INTERCEPT_TYPE_X64_IO_PORT
typedef UINT16 HV_X64_IO_PORT;

typedef union
{
      // Ensure that the parameter is 8 bytes wide
      UINT64      AsUINT64;

      // HvInterceptTypeX64IoPort
      HV_X64_IO_PORT    IoPort;

      // HvInterceptTypeX64Msr has no parameters

      // HvInterceptTypeX64Cpuid
      UINT32      CpuidIndex;

      // HvInterceptTypeX64Exception
      UINT16      ExceptionVector;} HV_INTERCEPT_PARAMETERS;

typedef struct
{
      HV_INTERCEPT_TYPE Type;
      HV_INTERCEPT_PARAMETERS Parameters;
} HV_INTERCEPT_DESCRIPTOR;
```

9.2.3 Intercept Access Types

This data type is used to specify different access types that can trigger intercepts. A corresponding mask can be used to specify combinations of accesses (for example, reads and writes).

```
typedef UINT8 HV_INTERCEPT_ACCESS_TYPE_MASK;

#define HV_INTERCEPT_ACCESS_MASK_NONE     0
#define HV_INTERCEPT_ACCESS_MASK_READ     1
#define HV_INTERCEPT_ACCESS_MASK_WRITE    2
#define HV_INTERCEPT_ACCESS_MASK_EXECUTE  4
```

9.2.4 Unsupported Feature Codes

The hypervisor reports an attempt to access an “unsupported feature” using an intercept message. The message field *FeatureCode* is used to indicate the specific feature that the guest attempted to use. The codes are defined as follows:

```
typedef enum
{
      HvUnsupportedFeatureIntercept       = 1,
HvUnsupportedFeatureTaskSwitchTss                 = 2
} HV_UNSUPPORTED_FEATURE_CODE;
```

9.3 Intercept Interfaces

9.3.1 HvInstallIntercept

The HvInstallIntercept hypercall enables or disables intercepts triggered by the specified child partition’s virtual processors.

Wrapper Interface

```
HV_STATUS
HvInstallIntercept(
      __in HV_PARTITION_ID    PartitionId,
      __in HV_INTERCEPT_ACCESS_TYPE_MASK  AccessType,
      __in PCHV_INTERCEPT_DESCRIPTOR      Descriptor
      );
```

Native Interface

HvInstallIntercept		
	Call Code = 0x004D	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	AccessType (4 bytes)	InterceptType (4 bytes)
16	InterceptParameter (8 bytes)	

Description

Intercepts are installed by specifying the appropriate access type as described in section 9.1.1. To clear intercepts, the caller should specify an access type of HV_INTERCEPT_ACCESS_MASK_NONE.

In general, calls to HvInstallIntercept can fail if the hypervisor has insufficient memory required to allocate the requisite internal data structures. However, some intercept types do not require dynamic memory allocation and are guaranteed to never fail with an insufficient-memory error.

Intercept Type	Can Fail Due to Insufficient Memory?
I/O port access	No
MSR access	No
CPUID instruction execution	Yes
Exceptions	No

Input Parameters

PartitionId specifies a child partition.

AccessType specifies the access type(s) that should be intercepted. An access type of "none" indicates that the intercept should be removed. Other combinations of access type flags are permitted depending on the intercept type.

InterceptType specifies the type of intercept being installed.

InterceptParameter provides a parameter whose meaning is specific to the intercept type.

Output Parameters

None.

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_PARAMETER	The specified intercept type is invalid.
	The specified access type is not permitted for the given intercept type. (For valid access types, see the previous table.)
	The specified intercept parameters are invalid or out of range for the given intercept type.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the partition's memory pool is insufficient to perform the operation.

9.4 Intercept Messages and Message Formats

When an intercepted event occurs in a child partition, the hypervisor informs the parent partition by using a message. For a detailed description of the messages sent by the hypervisor when a processor intercept is triggered, see chapter 16.

10 Virtual Processor Management

10.1 Overview

Each partition may have zero or more virtual processors. This section specifies how the state of a virtual processor can be programmatically accessed or modified by another sufficiently privileged partition by using hypercalls. This facility can be used in conjunction with intercepts to virtualize the processor.

10.1.1 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.

10.1.2 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.

10.1.3 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.

A suspended virtual processor consumes no processor cycles. It is stopped on a guest instruction boundary. A virtual processor is suspended *when any* of the following are true:

The virtual processor is explicitly suspended

The virtual processor is implicitly suspended due to an intercept

Both suspension reasons must be cleared before a virtual processor becomes active. They can be set or cleared by using HvSetVpRegisters.

10.1.4 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 in to a low power idle state to be woken with the arrival of an interrupt even when interrupt is masked on the virtual processor. 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 5.2.3) 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 if the interrupt is enabled on the virtual processor or not.

10.1.5 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.

10.1.6 Virtual Processor APIC IDs

On a real x64 system, each processor starts with a hard-coded *initial APIC ID*. This value can be retrieved through the CPUID instruction. On some systems, the initial APIC ID is not necessarily unique across all processors, so the APIC ID accessible through the APIC's MMIO space can be modified. This allows software to allocate unique APIC IDs for all processors within the system.

Virtual processors also have an initial APIC ID value that cannot be modified by software running within the partition. This value is initialized at virtual processor creation time to the virtual processor's index. The value can be modified by the parent partition by calling HvSetVpRegisters and specifying the register HvX64RegisterInitialApicId.

Virtual processors also have a guest-programmable APIC ID that is accessible through the APIC's MMIO space. This value is also initialized based on the virtual processor's index.

10.2 Virtual Processor Data Types

10.2.1 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       0xFFFFFFFF
```

A virtual processor's ID can be retrieved by the guest through a hypervisor-defined MSR (model-specific register) HV_X64_MSR_VP_INDEX.

```
#define HV_X64_MSR_VP_INDEX 0x40000002
```

10.2.2 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,

    // Interrupt Registers
    HvX64RegisterPendingInterruption    = 0x00010002,
    HvX64RegisterInterruptState   = 0x00010003,

    // 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,

    // 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,

    HvX64RegisterMtrrPhysMask0    = 0x00080040,
    HvX64RegisterMtrrPhysMask1    = 0x00080041,
    HvX64RegisterMtrrPhysMask2    = 0x00080042,
    HvX64RegisterMtrrPhysMask3    = 0x00080043,
    HvX64RegisterMtrrPhysMask4    = 0x00080044,
    HvX64RegisterMtrrPhysMask5    = 0x00080045,
    HvX64RegisterMtrrPhysMask6    = 0x00080046,
    HvX64RegisterMtrrPhysMask7    = 0x00080047,

    HvX64RegisterMtrrFix64k00000  = 0x00080070,
    HvX64RegisterMtrrFix16k80000  = 0x00080071,
    HvX64RegisterMtrrFix16kA0000  = 0x00080072,
    HvX64RegisterMtrrFix4kC0000   = 0x00080073,
    HvX64RegisterMtrrFix4kC8000   = 0x00080074,
    HvX64RegisterMtrrFix4kD0000   = 0x00080075,
```

```
    HvX64RegisterMtrrFix4kD8000   = 0x00080076,
    HvX64RegisterMtrrFix4kE0000   = 0x00080077,
    HvX64RegisterMtrrFix4kE8000   = 0x00080078,
    HvX64RegisterMtrrFix4kF0000   = 0x00080079,
    HvX64RegisterMtrrFix4kF8000   = 0x0008007A,

    // Hypervisor-defined MSRs (Misc)
    HvX64RegisterVpRuntime  = 0x00090000,
    HvX64RegisterHypercall  = 0x00090001,
    HvX64RegisterGuestOsId  = 0x00090002,
    HvX64RegisterVpIndex    = 0x00090003,
    HvX64RegisterTimeRefCount     = 0x00090004,

    // Virtual APIC registers MSRs
    HvX64RegisterEoi                    = 0x00090010,
    HvX64RegisterIcr                    = 0x00090011,
    HvX64RegisterTpr                    = 0x00090012,
    HvX64RegisterApicAssistPage         = 0x00090013,

    // Performance statistics MSRs
    HvX64RegisterStatsPartitionRetail   = 0x00090020,
    HvX64RegisterStatsPartitionInternal = 0x00090021,
    HvX64RegisterStatsVpRetail          = 0x00090022,
    HvX64RegisterStatsVpInternal        = 0x00090023,


    // Hypervisor-defined MSRs (Synic)
    HvX64RegisterSint0      = 0x000A0000,
    HvX64RegisterSint1      = 0x000A0001,
    HvX64RegisterSint2      = 0x000A0002,
    HvX64RegisterSint3      = 0x000A0003,
    HvX64RegisterSint4      = 0x000A0004,
    HvX64RegisterSint5      = 0x000A0005,
    HvX64RegisterSint6      = 0x000A0006,
    HvX64RegisterSint7      = 0x000A0007,
    HvX64RegisterSint8      = 0x000A0008,
    HvX64RegisterSint9      = 0x000A0009,
    HvX64RegisterSint10     = 0x000A000A,
    HvX64RegisterSint11     = 0x000A000B,
    HvX64RegisterSint12     = 0x000A000C,
    HvX64RegisterSint13     = 0x000A000D,
    HvX64RegisterSint14     = 0x000A000E,
    HvX64RegisterSint15     = 0x000A000F,
    HvX64RegisterScontrol   = 0x000A0010,
    HvX64RegisterSversion   = 0x000A0011,
    HvX64RegisterSiefp      = 0x000A0012,
    HvX64RegisterSimp = 0x000A0013,
    HvX64RegisterEom  = 0x000A0014,

    // Hypervisor-defined MSRs (Synthetic Timers)
    HvX64RegisterStimer0Config    = 0x000B0000,
    HvX64RegisterStimer0Count     = 0x000B0001,
    HvX64RegisterStimer1Config    = 0x000B0002,
    HvX64RegisterStimer1Count     = 0x000B0003,
    HvX64RegisterStimer2Config    = 0x000B0004,
    HvX64RegisterStimer2Count     = 0x000B0005,
    HvX64RegisterStimer3Config    = 0x000B0006,
    HvX64RegisterStimer3Count     = 0x000B0007,

    //
  // 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

} HV_REGISTER_NAME;
```

10.2.3 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_REGISTER_VALUE;

typedef HV_REGISTER_VALUE *PHV_REGISTER_VALUE;
```

10.3 Virtual Processor Register Formats

10.3.1 Virtual Processor Suspend Registers

The hypervisor recognizes two registers that allow the virtual processor to be suspended or unsuspended. Two registers are specified so callers can easily set or clear the two values independently. Unlike most registers, these are not defined by the underlying processor architecture and cannot be accessed by the virtual processor they are associated with.

The first register (HvRegisterExplicitSuspend) is used for cases where the virtual processor is to be explicitly suspended using the HvSetVpRegisters hypercall.

```
typedef struct
{
      UINT64      Suspended:1;
      UINT64      Reserved:63;
} HV_EXPLICIT_SUSPEND_REGISTER;
```

The second register (HvRegisterInterceptSuspend) is set when a virtual processor is suspended by the hypervisor due to an intercept. Bits that are set in this register may be cleared together or independently using the HvSetVpRegisters hypercall. Note that bits that are clear may only be set by the hypervisor and any attempt by a guest to set them will result in an error. For more information about suspending a virtual processor, see section 10.1.3.

This register also contains a bit used to inhibit the virtual processor's TLB from being flushed from another processor using the HvFlushVirtualAddressSpace or HvFlushVirtualAddressList hypercalls. If such an attempt is made while the flush inhibit bit is set, the caller's virtual processor will be suspended. When the flush inhibit bit is cleared, all other virtual processors suspended by an attempt to flush that virtual processor's TLB will be unsuspended. For more information about TLB flush restrictions, see section 12.1.4.

```
typedef struct
{
      UINT64      Suspended:1;
      UINT64      TlbFlushInhibit:1;
      UINT64      Reserved:62;
} HV_INTERCEPT_SUSPEND_REGISTER;
```

10.3.2 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

10.3.3 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;
```

10.3.4 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
} 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      Reserved1:11;
      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.

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.

10.3.5 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 IgnNe:1;
      UINT8 Reserved:7;
      UINT16      LastFpOp;
      union
      {
            UINT64      LastFpRip;
            struct
            {
                  UINT32      LastFpEip;
                  UINT16      LastFpCs;
            };
      };
} HV_X64_FP_STATUS_CONTROL_REGISTER;

typedef struct
{
      union
      {
            UINT64      LastFpRdp;
            struct
            {
                  UINT32      LastFpDp;
                  UINT16      LastFpDs;
            };
      };
      UINT32      XmmStatusControl;
      UINT32      XmmStatusControlMask;
} HV_X64_XMM_STATUS_CONTROL_REGISTER;
```

10.3.6 Virtual Processor Segment Registers

Segment register state is encoded as follows:

```
typedef struct
{
      UINT64      Base;
      UINT32      Limit;
      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;
      };
      UINT16      Selector;
} 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.

10.3.7 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;
```

10.4 Virtual Processor Interfaces

10.4.1 HvCreateVp

The HvCreateVp hypercall creates a new virtual processor in a partition.

Wrapper Interface

```
HV_STATUS
HvCreateVp(
      __in HV_PARTITION_ID                PartitionId,
      __in HV_PROXIMITY_DOMAIN_INFO       ProximityDomainInfo,
      __in HV_VP_INDEX              VpIndex,
      __in UINT64             Flags
      );
```

Native Interface

HvCreateVp		
	Call Code = 0x004E	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	VpIndex (4 bytes)	Padding (4 bytes)
16	ProximityDomainInfo (8 bytes)	
24	Flags (8 bytes)	

Description

The specified virtual processor index must be below the maximum virtual processor index supported by the hypervisor implementation.

When a virtual processor is successfully created, its architected state is initialized consistent with the reset state of a real processor. The initial APIC ID is set to the specified virtual processor index. The parent partition can modify this value by calling HvSetRegisterState and specifying the register HvX64RegisterInitialApicId. The guest-programmable APIC ID, which is accessible through the APIC's MMIO space, is also initialized based on the specified virtual processor index.

Upon creation, a virtual processor is marked suspended (that is, HvRegisterExplicitSuspend contains a value of 1). This value must be cleared before the virtual processor will begin executing instructions.

A virtual processor created with an index of zero will be designated as the *boot processor* and placed in the ready state. The virtual processor will begin executing instructions once its HvExplicitSuspendRegister is cleared. Virtual processors created with a non-zero index will be *application processors* and placed in the waiting state. Such processors must receive a SIPI interrupt (*HvX64InterruptTypeSipi)* in order to begin execution.

Input Parameters

PartitionId specifies the partition.

VpIndex specifies the index of the virtual processor.

ProximityDomainInfo specifies the ACPI proximity domain information of the NUMA node where the virtual processor's initial data structures will reside. If there are no pages in the caller's pool for the specified ID, then the call will fail. The Proximity Domain specifier is described in section 7.2.1.

Flags is an options mask that affects the creation of the virtual processor. No flags are currently defined and this parameter must be zero.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition.
HV_STATUS_OPERATION_DENIED	The partition has per-VP reserve or capacity property value defined and creation of a virtual processor would cause the total reserve or capacity to exceed 100%.
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	The specified flags value is not zero.
	The *ProximityDomainInfo* parameter specifies an invalid flag bit or an invalid domain ID.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INSUFFICIENT_MEMORY	The memory pool of the specified partition does not contain sufficient pages to perform the operation.
HV_STATUS_NO_RESOURCES	A required system resource is unavailable or an implementation limit has been reached.

10.4.2 HvDeleteVp

The HvDeleteVp hypercall deletes an existing virtual processor and its related data structures.

Wrapper Interface

```
HV_STATUS
HvDeleteVp(
      __in HV_PARTITION_ID    PartitionId,
      __in HV_VP_INDEX  VpIndex
      );
```

Native Interface

HvDeleteVp [fast]		
	Call Code = 0x004F	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	VpIndex (4 bytes)	Padding (4 bytes)

Description

It returns the memory consumed by its internal data structures back to the memory pool of its partition.

Input Parameters

PartitionId specifies the partition.

VpIndex specifies the index of the virtual processor.

Output Parameters

None.

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 is invalid.
HV_STATUS_INVALID_VP_INDEX	The specified VP index does not reference a virtual processor within the specified partition.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

10.4.3 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,
      __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)	Padding (4 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.

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.

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.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

10.4.4 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,
      __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)	RsvdZ (4 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.

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 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 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.
	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.

11 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.

Some differences exist between AMD's and Intel's implementations. These differences, which are visible to guests, may include the following. Note that this list is not comprehensive and is likely to change as newer generations of processors are released:

Intel processors support:

- SSE3 instruction
- Hyperthreading

AMD processors support:

- 3DNow! instructions
- Fast FXSAVE mechanism
- Additional MMX extensions
- PREFETCHW instruction
- SAHF and LAHF instructions in long mode

11.1 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.

11.2 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.

11.3 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. It will present all other partitions with the value obtained from the logical processor possessing the minimum family, model and stepping value.

11.4 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.

11.5 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 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 3.4). Any attempt to use these instructions when the hypervisor does not indicate availability will result in a #UD fault.

11.6 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.

11.7 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.

11.8 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
}
```

11.9 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
}
```

11.10 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 27

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. The tables in 29 outline the policy for those MSRs whose contents are modified by the hypervisor.

11.10.1 Modified MSR Behavior

Certain MSR accesses are intercepted by the hypervisor. The behavior of MSR accesses by the root guest may differ from accesses by other guests. These are described in Appendix F: Architectural MSRs.

11.11 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. The table in Appendix D: Architectural CPUID details the ways in which CPUID information is modified by the hypervisor. In some cases, the CPUID values returned to the root partition differ from non-root partitions. In such cases, the differences are noted.

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.

11.12 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.

The following pseudo-code shows what occurs when an exception occurs.

```
if an intercept has been set for the exception
{
      Suspend VP and send message to parent (Exception Intercept)
}
else
{
      Exception is generated normally
}
```

12 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.

12.1 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.

12.1.1 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.
- 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 SPA pages by other partitions or by devices (that is, through DMA) may not trigger the appropriate TLB flush.
- Internally, the hypervisor may use *shadow page tables* that translate GVAs 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.

12.1.2 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).

12.1.3 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.
- *UseHypercallForRemoteFlush*: 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.

12.1.4 Restrictions on TLB Flushes

When a virtual processor generates an intercept—especially those associated with memory accesses, software running within the parent 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.

12.2 Memory Cache Control Overview

12.2.1 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.

12.2.2 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.

12.3 Virtual MMU Data Types

12.3.1 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;
```

12.3.2 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
```

12.3.3 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;
```

12.3.4 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
```

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
} 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;
```

12.3.5 Gpa Access Types

The calls HvReadGpa and HvWriteGpa take a collection of input control flags and return a result code. The input control flags are defined as follows:

```
typedef struct
{
      UINT64      CacheType:8;      // Cache type for access
      UINT64      Reserved:56;
} HV_ACCESS_GPA_CONTROL_FLAGS;
```

The return result code is defined as follows:

```
typedef enum
{
      HvAccessGpaSuccess            = 0,

      // GPA access failures
      HvAccessGpaUnmapped           = 1,
      HvAccessGpaReadIntercept      = 2,
      HvAccessGpaWriteIntercept     = 3,
      HvAccessGpaIllegalOverlayAccess     = 4
} HV_ACCESS_GPA_RESULT_CODE;

typedef struct
{
      HV_ACCESS_GPA_RESULT_CODE     ResultCode;
      UINT32      Reserved;
} HV_ACCESS_GPA_RESULT;

typedef HV_ACCESS_GPA_RESULT *PHV_ACCESS_GPA_RESULT;
```

12.4 Virtual MMU Interfaces

12.4.1 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 (for restrictions, see earlier in this specification).
	One or more reserved bits in the specified address space ID (as defined by the x64 architecture) were set.

12.4.2 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.

NOTE

Future versions of the hypervisor may support more than 64 virtual processors per partition. In that case, a new field will be added to the flags value that allows the caller to define the “processor bank” to which the processor mask applies.

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. For more information on TLB locking, see section 12.1.4.

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.

12.4.3 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. For more information on TLB flush inhibit, see section 12.1.4.

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.

12.4.4 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_LOCK_TLB: 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 TLB of the specified virtual processor until after the *TlbFlushInhibit* flag is cleared. See section 12.1.4 for more information.

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.

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.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.

12.4.5 HvReadGpa

The HvReadGpa hypercall attempts to read from a range of bytes within the specified GPA page.

Wrapper Interface

```
HV_STATUS
HvReadGpa(
      __in  HV_PARTITION_ID   PartitionId,
      __in  HV_VP_INDEX VpIndex,
      __in  HV_GPA      BaseGpa,
      __in  UINT32      ByteCount,
      __in  HV_ACCESS_GPA_CONTROL_FLAGS   ControlFlags,
      __out PHV_ACCESS_GPA_RESULT   AccessResult,
      __out_ecount(ByteCount) PVOID DataBuffer
      );
```

Native Interface

HvReadGpa		
	Call Code = 0x0053	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	VpIndex (4 bytes)	ByteCount (4 bytes)
16	BaseGpa (8 bytes)	
24	ControlFlags (8 bytes)	
⬅ Output Parameters		
0	AccessResult (8 bytes)	
8	DataLow (8 bytes)	
16	DataHigh (8 bytes)	

Description

The range of bytes being read from is not allowed to cross a page boundary. Up to 16 bytes of data can be read. A byte count of zero is invalid.

The hypervisor performs the access according to the current GPA mapping. If the specified GPA references an overlay page, the access is directed to the overlay page and not the page that is potentially mapped to the specified GPA. This is also true of MMIO ranges managed by the hypervisor (namely, the APIC page). In other words, the access is performed as though the virtual processor itself performed the access.

Data will be accessed in an aligned manner using the appropriate combination of 32-bit, 16-bit and 8-bit reads and will be performed in ascending address order.

Accesses to overlay pages perform the side effects normally associated with the access.

Atomicity of the access is not guaranteed by this call. Another virtual processor may be attempting to modify the specified GPA range. Unless the other virtual processor uses atomic accesses, the call may return partially-modified data.

If the call returns HV_STATUS_SUCCESS, the *AccessResult* parameter is valid. Callers must check the *AccessResult* field to determine whether an error was detected when the specified GPA page was accessed. This can occur for one of three reasons:

The GPA is unmapped. In this case, *HvAccessGpaUnmapped* is returned.

The GPA mapping's access rights indicate that the page is not readable. In this case, *HvAccessGpaReadIntercept* is returned.

The access targets an overlay page that doesn't allow reads. In this case, *HvAccessGpaWriteIntercept* is returned.

Input Parameters

PartitionId specifies a partition.

VpIndex specifies a virtual processor index.

BaseGpa specifies the first guest physical address to be accessed.

ByteCount specifies the number of bytes to access (1 through 16, inclusive).

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

Output Parameters

AccessResult specifies information about the access including the result code.

Data returns the data that was read. *DataLow* contains bytes 1-8 and *DataHigh* contains bytes 9-16. *If* the caller specified a byte count smaller than 16 bytes, the remaining bytes in the output data will contain zeroed bytes.

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	The specified GPA is beyond the end of the partition's GPA space.
	The specified access crosses a page boundary.
	The specified byte count is 0 or greater than 16.
	A reserved bit within the control flags is set.
	The specified cache type is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

12.4.6 HvWriteGpa

The HvWriteGpa hypercall attempts to write to a range of bytes within the specified GPA page.

Wrapper Interface

```
HV_STATUS
HvWriteGpa(
      __in  HV_PARTITION_ID   PartitionId,
      __in  HV_VP_INDEX VpIndex,
      __in  HV_GPA      BaseGpa,
      __in  UINT32      ByteCount,
      __in  HV_ACCESS_GPA_CONTROL_FLAGS   ControlFlags,
      __in_ecount(ByteCount) PCVOID DataBuffer,
      __out PHV_ACCESS_GPA_RESULT   AccessResult
      );
```

Native Interface

HvWriteGpa		
	Call Code = 0x0054	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	VpIndex (4 bytes)	ByteCount (4 bytes)
16	BaseGpa (8 bytes)	
24	ControlFlags (8 bytes)	
32	DataLow (8 bytes)	
40	DataHigh (8 bytes)	
⬅ Output Parameters		
0	AccessResult (8 bytes)	

Description

The range of bytes being written to is not allowed to cross a page boundary. Up to 16 bytes of data can be written. A byte count of zero is invalid.

The hypervisor performs the access according to the current GPA mapping. If the specified GPA references an overlay page, the access is directed to the overlay page and not the page that is potentially mapped to the specified GPA. This is also true of MMIO ranges managed by the hypervisor (namely, the APIC page). In other words, the access is performed as though the virtual processor itself performed the access.

Data will be accessed in an aligned manner using the appropriate combination of 32-bit, 16-bit and 8-bit writes and will be performed in ascending address order.

Accesses to overlay pages perform the side effects normally associated with the access.

Atomicity of the access is not guaranteed by this call. Another virtual processor may be attempting to read or write specified GPA range. Unless the other virtual processor uses atomic accesses, the other virtual processor may observe partially-written data by this call.

If the call returns HV_STATUS_SUCCESS, the *AccessResult* parameter is valid. Callers must check the *AccessResult* field to determine whether an error was detected when the specified GPA page was accessed. This can occur for one of three reasons:

The GPA is unmapped. In this case, *HvAccessGpaUnmapped* is returned.

The GPA mapping's access rights indicate that the page is not writable. In this case, *HvAccessGpaReadIntercept* is returned.

The access targets an overlay page that doesn't allow writes. In this case, *HvAccessGpaWriteIntercept* is returned.

Input Parameters

PartitionId specifies a partition.

VpIndex specifies a virtual processor index.

BaseGpa specifies the first guest physical address to be accessed.

ByteCount specifies the number of bytes to access (1 through 16, inclusive).

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

Data specifies the data to be written. *DataLow* contains bytes 1-8 and *DataHigh* contains bytes 9-16. If the caller specified a byte count smaller than 16 bytes, the remaining bytes in the input data are ignored.

Output Parameters

AccessResult specifies information about the access including the result code.

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	The specified GPA is beyond the end of the partition’s GPA space.
	The specified access crosses a page boundary.
	The specified byte count is 0 or greater than 16.
	A reserved bit within the control flags is set.
	The specified cache type is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.

13 Virtual Interrupt Control

13.1 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 (in which case the interrupt is reflected to the root partition).
- A parent partition calls HvAssertVirtualInterrupt (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 calls HvPostMessage.
- Another partition calls HvSignalEvent.

Internal interrupts are generated in the following situations:

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

13.2 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.

13.2.1 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 13.2.2. 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 remaining parts of this section describe only those aspects of SynIC functionality that are extensions of the local APIC.

13.2.2 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.

13.2.3 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 section 3.4)

13.2.3.1 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.

13.2.3.2 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.

13.2.3.3 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.

13.3 Virtual Interrupts

13.3.1 Virtual Interrupt Overview

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

13.3.2 Virtual Interrupt Types

To send a virtual interrupt, software (typically running within the parent partition) must call HvAssertVirtualInterrupt and specify a virtual processor within the target partition. 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. This type is also special in that it is always directed at the boot processor. It also requires the use of a separate hypercall, HvClearVirtualInterrupt, to clear an acknowledged interrupt before subsequent interrupts of this type can be asserted.

13.3.3 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	??

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.

13.3.4 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 EOIed, 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.

For more information on intercepts, see section 9.1.1.

13.3.4.1 APIC Assist Page Register

The hypervisor provides a virtual APIC assist page per virtual processor which is overlaid on the guest GPA space. The OS has read/write access to the virtual APIC assist page. It specifies the location of the overlay page (in GPA space) by writing to the Virtual APIC Assist MSR (0x40000073). The format of the Virtual APIC Assist Page MSR is as follows:

63:12	11:1	0
Virtual APIC Assist Page Base Address	RsvdP	Enable

Currently the only defined field in the virtual APIC assist page is the EOI Assist field. The EOI Assist field resides at offset 0 of the overlay page and is DWORD sized. 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 APIC 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.

13.4 Virtual Interrupt Data Types

13.4.1 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;
```

13.4.2 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;
```

13.4.3 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
```

13.5 Virtual Interrupt Interfaces

13.5.1 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
      );
```

Native Interface

HvAssertVirtualInterrupt		
	Call Code = 0x0055	
➡ Input Parameters		
0	DestinationPartition (8 bytes)	
8	InterruptControl (8 bytes)	
16	DestinationAddress (8 bytes)	
24	RequestedVector (4 bytes)	Padding (4 bytes)

Description

For information on virtual interrupts, see section 13.2.3.

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.

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.
	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.

13.5.2 HvClearVirtualInterrupt

The HvClearVirtualInterrupt hypercall determines whether a previously-requested virtual interrupt of type *HvX64InterruptTypeExtInt* has been acknowledged.

Wrapper Interface

```
HV_STATUS
HvClearVirtualInterrupt(
      __in  HV_PARTITION_ID   DestinationPartition
      );
```

Native Interface

HvClearVirtualInterrupt [fast]	
	Call Code = 0x0056
➡ Input Parameters	
0	DestinationPartition (8 bytes)

Description

An external interrupt is considered “acknowledged” when it has been delivered to the boot processor.

The call clears the acknowledged state which allows subsequent external interrupts to be asserted. If no external interrupt has been acknowledged, then the call fails with a status code of HV_STATUS_NOT_ACKNOWLEDGED.

For more information on virtual interrupts, see section 13.2.3

Input Parameters

DestinationPartition specifies the partition.

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_NOT_ACKNOWLEDGED	An external interrupt has not been asserted and acknowledged since the last call to HvClearVirtualInterrupt.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

14 Inter-Partition Communication

14.1 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).

14.2 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.

14.2.1 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:

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 (see section 14.9.1). These message buffers are returned to the memory pool when the port is deleted (see section 14.9.2).

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. For more information about timers, see chapter 15, and for a detailed description of timer messages, see section 16.4.

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.

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. For details about event logging, see chapter 19.

14.2.2 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.

14.2.3 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 (see section 14.6.1) or disabling the SIMP (see section 14.6.4).

Breaking a connection (see HvDisconnectPort, section 14.9.4) will not affect undelivered (queued) messages. Deletion of the target port (see HvDeletePort, section 14.9.2) 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.

14.2.4 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 14.7) 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 14.7 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.

14.2.5 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.

14.2.6 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 mechanism defined in chapter 15 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. Timer messages are defined in section 16.4.

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.

14.3 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.

14.3.1 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 14.7 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).

14.3.2 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.

14.3.3 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.

14.4 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.

14.5 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.

14.5.1 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.

14.5.2 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.

14.5.3 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.

14.5.4 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.

14.6 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

MSR Address	Register Name	Function
0x4000009F	SINT15	Interrupt source 15

14.6.1 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 (see chapter 14 for details). 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.

14.6.2 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.

14.6.3 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.

14.6.4 SIMP Register

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

Bits	Description	Attributes

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.

14.6.5 SINTx Registers

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

Bits	Description	Attributes
63:18	RsvdP (value should be preserved)	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.

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.

14.6.6 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.

14.7 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 section 8.1.3.

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.

14.8 Inter-Partition Communication Data Types

14.8.1 Synthetic Interrupt Sources

The SynIC supports 16 synthetic interrupt sources.

```
#define HV_SYNIC_SINT_COUNT   16
```

```
typedef UINT32 HV_SYNIC_SINT_INDEX;
```

14.8.2 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,

      // Default (non-programmable) intercept messages
      HvMessageTypeApicEoi    = 0x80000030,
      HvMessageTypeFerrAsserted     = 0x80000031,

      // Trace buffer messages
      HvMessageTypeEventLogBuffersComplete      = 0x80000040,

      // Platform-specific processor intercept messages
      HvMessageTypeX64IoPortIntercept     = 0x80010000,
      HvMessageTypeX64MsrIntercept  = 0x80010001,
      HvMessageTypeX64CpuidIntercept      = 0x80010002,
      HvMessageTypeX64ExceptionIntercept  = 0x80010003
} 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.

For a complete list of messages sent by the hypervisor, see section 16.2.

14.8.3 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;
```

14.8.4 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;
```

For a detailed description of the messages sent by the hypervisor, see chapter 16.

14.8.5 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;
```

14.8.6 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;
```

14.8.7 Port Properties

Port properties provide a generic way to set or obtain certain characteristics of a port. Properties are identified by a 32-bit code. The associated property value is 64 bits in size.

```
typedef UINT64 HV_PORT_PROPERTY;
typedef HV_PORT_PROPERTY *PHV_PORT_PROPERTY;

typedef enum
{
    HvPortPropertyPostCount                     = 0x00000000,
    HvPortPropertyPreferredTargetVp             = 0x00000001
} HV_PORT_PROPERTY_CODE;
```

The following table explains the meaning of each property.

Port property	Meaning
HvPortPropertyPostCount	The number of messages that have been successfully posted to the port since its creation.
HvPortPropertyPreferredTargetVp	Specifies the preferred virtual processor that will be targeted by messages received through this port. For ports created with *TargetVp* specified as HV_ANY_VP, HvPortPropertyPreferredTargetVp allows the caller to supply the preferred virtual processor locality.

14.8.8 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.

14.8.9 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;
```

14.8.9.1 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 1in 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.*

14.8.9.2 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.

14.8.10 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                  MonitorDisabled:1;
            UINT32                  ReservedZ:27;
      };
} HV_MONITOR_TRIGGER_STATE, *PHV_MONITOR_TRIGGER_STATE;
```

GroupEnable and MonitorDisabled are 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* and *MonitorDisabled* 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.

The *MonitorDisabled* flag is set when the hypervisor is temporarily not monitoring the monitor page. If the caller observes this condition, it may trigger the read of the page by calling the HVSignalEvent hypercall.

TriggerGroup is an array of four trigger group structures. For details, see section 14.8.9.1.

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 14.8.9.2 .

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

14.9 Inter-Partition Communication Interfaces

14.9.1 HvCreatePort

The HvCreatePort hypercall allocates a port object and a set of sixteen (16) message buffers to store queued pending messages.

Wrapper Interface

```
HV_STATUS
HvCreatePort(
      __in HV_PARTITION_ID    PortPartition,
      __in HV_PORT_ID   PortId,
      __in HV_PARTITION_ID    ConnectionPartition,
      __in PCHV_PORT_INFO     PortInfo
      );
```

Native Interface

HvCreatePort		
	Call Code = 0x0057	
➡ Input Parameters		
0	PortPartition (8 bytes)	
8	PortId (4 bytes)	Padding (4 bytes)
16	ConnectionPartition (8 bytes)	
24	PortTypeInfo (24 bytes)	
32		
40		

Description

The memory for the port object and the message buffers is allocated from the memory pool of the partition identified by the *PortPartition* parameter.

The specified port ID must be unique among all ports associated with the receive partition.

The *ConnectionPartition* parameter must be specified to prevent other partitions from maliciously connecting to the newly-created port. Only connections from specified partition will be allowed.

Input Parameters

PortPartition specifies the partition that is to receive messages or events through this port.

PortId specifies an ID that must be unique among all port IDs for this partition. It is used to refer to the port. It is also used to identify the source of a message when it arrives.

ConnectionPartition specifies the partition that will be allowed to send messages or events through this port.

PortTypeInfo specifies information about the port:

For message ports (*PortType* is HvPortTypeMessage), *PortTypeInfo* is as follows:

24	PortType (4 bytes) = HvPortTypeMessage	Padding (4 bytes)

32	TargetSint (4 bytes)	TargetVp (4 bytes)
40	RsvdZ (8 bytes)	

PortType specifies HvPortTypeMessage, indicating that this is a message port.

TargetSint specifies the synthetic interrupt source to use for this port and may be in the range of 1 through 15. SINT0 is reserved for hypervisor use.

TargetVp specifies the virtual processor that will be targeted by messages received through this port. In most cases, a value of HV_ANY_VP is recommended. This instructs the hypervisor to route the message to any virtual processor in the target partition.

For event ports (*PortType* is HvPortTypeEvent), *PortTypeInfo* is as follows:

24	PortType (4 bytes) = HvPortTypeEvent		Padding (4 bytes)
32	TargetSint (4 bytes)		TargetVp (4 bytes)
40	BaseFlagNumber (2 bytes)	FlagCount (2 bytes)	RsvdZ (4 bytes)

PortType specifies HvPortTypeEvent, indicating that this is an event signal port.

TargetSint specifies the synthetic interrupt source to use for this port and may be in the range of 1 through 15. SINT0 is reserved for hypervisor use.

TargetVp specifies the virtual processor that will be targeted by events received through this port. In most cases, a value of HV_ANY_VP is recommended. This instructs the hypervisor to route the event signal to any virtual processor in the target partition.

BaseFlagNumber specifies the starting event flag number to be assigned to the port and may be in the range of 0 through 2047.

FlagCount specifies the number of event flags, starting at *BaseFlagNumber*, which are to be assigned to the port and cannot be greater than 2048.

For monitor ports (*PortType* is HvPortTypeMonitor), *PortTypeInfo* is as follows:

24	PortType (4 bytes) = HvPortTypeMonitor	Padding (4 bytes)
32	MonitorAddress (8 bytes)	
40	RsvdZ (8 bytes)	

PortType specifies HvPortTypeMonitor, indicating that this is a monitor port.

MonitorAddress specifies the GPA where the monitor overlay page is to be mapped within the port partition.

Output Parameters

None.

Restrictions

The caller must possess the *CreatePort* privilege.

The partitions specified by *PortPartition* and *ConnectionPartition* must be in the “active” state.

The caller must be the parent of the partition specified by *PortPartition* or the caller must possess the *CreatePort* privilege and specify its own partition ID.

Return Values

Status code	Error condition

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified *PortPartition.* 2. The caller is the specified *PortPartition,* and the partition has the CreatePort partition privilege.
HV_STATUS_INVALID_PARTITION_ID	The specified port partition ID is invalid.
	The specified connection partition ID is invalid.
	The connection and port partition IDs reference the same partition.
HV_STATUS_INVALID_PORT_ID	The specified port ID is already in use.
	The reserved portion (high 8 bits) of the *PortId* are non-zero.
HV_INVALID_VP_INDEX	The specified virtual processor index does not reference a valid VP and is not HV_ANY_VP.
HV_STATUS_INVALID_PARAMETER	The specified interrupt source index is not in the range 1 through 15 inclusive.
	The specified port type is invalid.
	For event ports, the specified base flag number plus the flag count is greater than or equal to HV_EVENT_FLAGS_COUNT.
	For event ports, the specified flag count is zero.
	The reserved fields of the *PortTypeInfo* parameter are not set to zero.
HV_STATUS_INVALID_ALIGNMENT	The specified monitor overlay page GPA is not page aligned.
HV_STATUS_INVALID_PARTITION_STATE	The specified port partition is not in the "active" state.
	The specified connection partition is not in the "active" state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the caller is insufficient to perform the operation.
HV_STATUS_NO_RESOURCES	A required system resource is unavailable or an implementation limit has been reached.

14.9.2 HvDeletePort

The HvDeletePort hypercall deallocates a port object and the message buffers associated with it.

Wrapper Interface

```
HV_STATUS
HvDeletePort(
      __in HV_PARTITION_ID    PortPartition,
      __in HV_PORT_ID   PortId
      );
```

Native Interface

HvDeletePort		
	Call Code = 0x0058	
➡ Input Parameters		
0	PortPartition (8 bytes)	
8	PortId (4 bytes)	Padding (4 bytes)

Description

At port deletion, any undelivered messages present in the queue will be lost. Subsequent references to the specified port ID will fail. Also, subsequent attempts to send messages or signal events through a connection to this port will fail.

Messages that arrived through the port may have already been copied to a virtual processor's SIM page. It is therefore possible for messages with the deleted port's ID to arrive after HvDeletePort has been called.

Input Parameters

PortPartition specifies the partition.

PortId specifies the ID of a port that was previously allocated by calling HvCreatePort.

Output Parameters

None.

Restrictions

The caller must possess the *CreatePort* privilege.

The partition specified by *PortPartition* must be in the "active" state.

The caller must be the parent of the partition specified by *PortPartition* or the caller must possess the CreatePort privilege and specify its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified partition. 2. The caller is the specified partition, and the partition has the CreatePort partition privilege.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_PORT_ID	The specified port ID is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

14.9.3 HvConnectPort

The HvConnectPort hypercall creates a connection to a previously-created port in another partition.

Wrapper Interface

```
HV_STATUS
HvConnectPort(
      __in HV_PARTITION_ID    ConnectionPartition,
      __in HV_CONNECTION_ID   ConnectionId,
      __in HV_PARTITION_ID    PortPartition,
      __in HV_PORT_ID   PortId,
      __in PCHV_CONNECTION_INFO     ConnectionInfo
      );
```

Native Interface

HvConnectPort		
	Call Code = 0x0059	
➡ Input Parameters		
0	ConnectionPartition (8 bytes)	
8	ConnectionId (4 bytes)	Padding (4 bytes)
16	PortPartition (8 bytes)	
24	PortId (4 bytes)	Padding (4 bytes)
32	ConnectionInfo (8 bytes)	

Description

The memory for the connection object is allocated from the connection partition's memory pool.

The specified connection ID must be unique among all connections associated with the connection partition.

Input Parameters

ConnectionPartition specifies the partition that is to send messages or signal events using the connection.

ConnectionId specifies an ID that is unique among all connection IDs for this partition. It is used to refer to the connection.

PortPartition specifies the partition whose port is being connected to.

PortId specifies the ID of a previously-created port within the *PortPartition*.

ConnectionInfo specifies information about the connection based upon the type of the target port (the port that the connection is to be associated with):

For message ports (*PortType* is HvPortTypeMessage), *ConnectionInfo* is as follows:

30	RsvdZ (8 bytes)

For event ports (*PortType* is HvPortTypeEvent), *ConnectionInfo* is as follows:

30	RsvdZ (8 bytes)

For monitor ports (*PortType* is HvPortTypeMonitor), *ConnectionInfo* is as follows:

30	MonitorAddress (8 bytes)

MonitorAddress specifies the monitor overlay page GPA within the connection partition.

Output Parameters

None.

Restrictions

The caller must possess the *ConnectPort* privilege.

The partitions specified by *ConnectionPartition* and *PortPartition* must be in the "active" state.

The caller must be the parent of the partition specified by *ConnectionPartition* or the caller must specify its own partition ID.

Return Values

Status code	Error condition

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified *ConnectionPartition.* 2. The caller is the specified *ConnectionPartition,* and the partition has the ConnectPort partition property.
HV_STATUS_INVALID_PARTITION_ID	The specified connection partition ID is invalid.
	The specified port partition ID is invalid.
	The *PortPartition* does not match the *ConnectionPartition* that the port was created with (see section 14.9.1).
HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is already in use.
	The reserved portion (high 8 bits) of the *ConnectionId* are non-zero.
HV_STATUS_INVALID_PORT_ID	The specified port ID does not exist.
	The *PortPartition* does not have a defined port with the specified port ID.
	The port specified by the port ID is already connected.
	The *PortPartition* does not match the *ConnectionPartition* that the port was created with (see section 14.9.1).
	The reserved portion (high 8 bits) of the *PortId* are non-zero.
HV_STATUS_INVALID_ALIGNMENT	The specified monitor overlay page GPA is not page aligned.
HV_STATUS_INVALID_PARAMETER	The *ConnectionInfo* parameter is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified port partition is not in the “active” state.
	The specified connection partition is not in the “active” state.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the caller is insufficient to perform the operation.
HV_STATUS_NO_RESOURCES	A required system resource is unavailable or an implementation limit has been reached.

14.9.4 HvGetPortProperty

The HvGetPortProperty hypercall allows an authorized guest to read a property of a port.

Wrapper Interface

```
HV_STATUS
HvGetPortProperty(
      __in  HV_PARTITION_ID   PartitionId,
      __in  HV_PORT_ID  PortId,
      __in  HV_PORT_PROPERTY_CODE   PropertyCode,
      __out PHV_PORT_PROPERTY PropertyValue
);
```

Native Interface

HvGetPortProperty		
	Call Code = 0x005A	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	PortId (4 bytes)	Padding (4 bytes)
16	PropertyCode (4 bytes)	Padding (4 bytes)
⬅ Output Parameters		
0	PropertyValue (8 bytes)	

Description

Port properties are described in section 14.8.7.

Input Parameters

PartitionId specifies the partition that owns the port.

PortId specifies a port.

PropertyCode specifies the property whose value the caller is interested in retrieving.

Output Parameters

PropertyValue returns the property value.

Restrictions

The caller must possess the *CreatePort* privilege.

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 caller must specify its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified target partition or the partition itself.
	The caller does not possess the CreatePort partition privilege.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_PORT_ID	The specified port ID is invalid.
HV_STATUS_UNKNOWN_PROPERTY	The specified property code is not a recognized property.

14.9.5 HvSetPortProperty

The HvSetPortProperty hypercall allows an authorized guest to set a property of a port.

Wrapper Interface

```
HV_STATUS
HvSetPortProperty(
      __in  HV_PARTITION_ID   PartitionId,
      __in  HV_PORT_ID  PortId,
      __in  HV_PORT_PROPERTY_CODE   PropertyCode,
      __in  HV_PORT_PROPERTY  PropertyValue,
      __out PHV_PORT_PROPERTY PropertyValue
);
```

Native Interface

	Call Code = 0x0070	
0	PartitionId (8 bytes)	
8	PortId (4 bytes)	Padding (4 bytes)
16	PropertyCode (4 bytes)	Padding (4 bytes)
24	PropertyValue (8 bytes)	
0	PropertyValue (8 bytes)	

Description

Port properties are described in section 14.8.7.

Input Parameters

PartitionId specifies the partition that owns the port.

PortId specifies a port.

PropertyCode specifies the property whose value the caller is interested in setting.

PropertyValue specifies the value to which to set the property.

Output Parameters

PropertyValue returns the property value.

Restrictions

The caller must possess the *CreatePort* privilege.

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 caller must specify its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified target partition or the partition itself.
	The caller does not possess the CreatePort partition privilege.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_PORT_ID	The specified port ID is invalid.
HV_STATUS_UNKNOWN_PROPERTY	The specified property code is not a recognized property.

14.9.6 HvDisconnectPort

The HvDisconnectPort hypercall severs a connection created by HvConnectPort.

Wrapper Interface

```
HV_STATUS
HvDisconnectPort(
      __in HV_PARTITION_ID    ConnectionPartition,
      __in HV_CONNECTION_ID   ConnectionId
      );
```

Native Interface

HvDisconnectPort [fast]		
	Call Code = 0x005B	
➡ Input Parameters		
0	ConnectionPartition (8 bytes)	
8	ConnectionId (4 bytes)	Padding (4 bytes)

Description

When severing a connection, any messages present in the port's message queue will be unaffected.

Input Parameters

ConnectionPartition specifies the partition.

ConnectionId specifies the connection ID.

Output Parameters

None.

Restrictions

The caller must possess the *ConnectPort* privilege.

The partition specified by *ConnectionPartition* must be in the "active" state.

The caller must be the parent of the partition specified by *ConnectionPartition* or the caller must possess the *ConnectPort* privilege and specify its own partition ID.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	Neither of the following is true: 1. The caller is the parent of the specified partition. 2. The caller is the specified partition, and the partition has the ConnectPort partition property.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is invalid.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

14.9.7 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 14.

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.

Status code	Error condition
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.
	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.

14.9.8 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.

15 Timers

15.1 Overview

15.1.1 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 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).

15.1.2 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.

15.1.3 Synthetic Timers

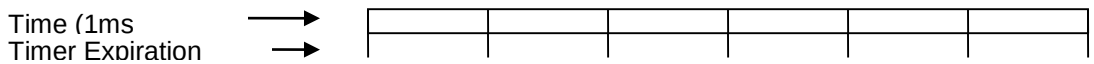
Synthetic timers provide a mechanism for generating an interrupt after some specified time in the future. Both one-shot and periodic timers are supported. A synthetic timer sends a message to a specified SynIC SINTx (synthetic interrupt source) upon expiration.

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.

15.1.4 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:

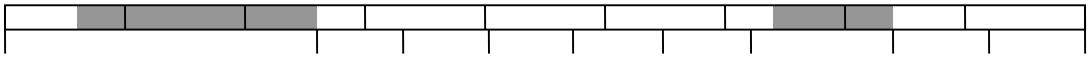


However, if the virtual processor used to signal the expiration is not available, some of the timer expirations may be delayed. A virtual processor may be unavailable because it is suspended (for example, during intercept handling) or because the hypervisor's scheduler decided that the virtual processor 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”.

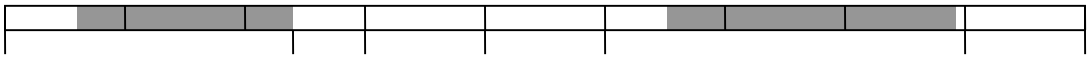
The following diagram shows the period modulation technique.



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.



15.1.5 Periodic Timer Assist

The parent partition may enable a periodic timer that is to be applied to a child partition. A period defines the rate of the interrupts, and a local APIC destination specifies which virtual processors are to be interrupted. The local APIC destination is usually defined by the I/O APIC redirection entry associated with the assisted timer source. The period is established by the parent partition by programming the periodic timer assist. Using this timer assist will reduce the cost of interrupt delivery and an improve interrupt accuracy.

The periodic timer assist is managed using two partition properties, HvPartitionPropertyEmulatedTimerPeriod and HvPartitionPropertyEmulatedTimerControl. Both properties default to all zeroes.

The timer period property specifies the time in 100 nanosecond units and is defined as follows:

```
// Timer Period
typedef union
{
      UINT64                  AsUINT64;
      HV_NANO100_DURATION     Period;
} HV_EMULATED_TIMER_PERIOD, *PHV_EMULATED_TIMER_PERIOD;
```

The local APIC destination is encoded in the timer control and is defined as follows:

```
// Periodic Timer Control
typedef union
{
    UINT64  AsUINT64;
      struct
      {
            UINT32      Vector:8;
            UINT32      DeliveryMode:3;
            UINT32      LogicalDestinationMode:1;
            UINT32      Enabled:1;
            UINT32      Reserved1:19;
            UINT32      Reserved2:24;
            UINT32      Mda[3]:8;
      };
} HV_EMULATED_TIMER_CONTROL, *PHV_EMULATED_TIMER_CONTROL;
```

The defined (named) fields of the Periodic Timer Control conform to those used by the local APIC's Interrupt Command Register (ICR). When the timer period property is set to zero, the timer control *Enabled* flag will be cleared, disabling the timer assist.

Normally, HvPartitionPropertyEmulatedTimerPeriod is set, and then HvPartitionPropertyEmulatedTimerControl updated to enable the assist. If those operations are done in the reverse order, the pending timer with the old timer period will be removed and a new timer with the new period inserted. Likewise, if the values in HvPartitionPropertyEmulatedTimerControl are changed without disabling the timer, the new values will take effect immediately.

The periodic timer assist acts in the same way as the non-lazy periodic timer described in the previous section, i.e. it will try to "catch up" if timer ticks are missed.

15.1.6 PM Timer Assist

The parent partition may request the hypervisor to assist the child's use of the ACPI PM timer. The parent specifies an I/O port and whether the child's ACPI PM Timer is 24- or 32-bits wide. When the child partition performs an `IN` from the specified port, the hypervisor will directly provide the value of the PM timer based upon the partition's reference time. The returned ACPI PM Timer value is computed as:

ACPI_PM_Timer = Modulus(Partition_Reference_Time * 0.3579545, 2**[24|32]);

This assist supports only `IN` operations on the I/O port (note that `INS` is not supported). Attempts to `OUT` to the port, to `IN` from any of the three subsequent I/O ports (that is, ports n+1, n+2 or n+3) or to IN from any of the three preceding I/O ports (that is, ports n-1, n-2 or n-3) will follow the rules for I/O port accesses as described in section 11.9.

If both an intercept and the PM timer assist are installed for the same I/O port, the assist will take precedence for the `IN` operation. The hypervisor will generate an intercept message for all `OUT` operations.

The caller specifies the PM Timer property's port and whether the value returned is 24 or 32 bits wide as follows:

[3] Mda is also known as MessageDestinationAddress.

```
// ACPI PM Timer
typedef union
{
      UINT64      AsUINT64;
      struct
      {
            UINT64      Port:16;
            UINT64      Width24:1;
            UINT64      Enabled:1;
            UINT64      ReservedZ1:14;
            UINT64      ReservedZ2:32;
      };
} HV_PM_TIMER_INFO, *PHV_PM_TIMER_INFO;
```

Port is the I/O port that the reference timer assist is to use.

Width24 indicates the width of the timer data. When set, the data is 24 bits wide. When clear, it is 32 bits wide.

Enabled is used to enable or disable the assist.

15.1.7 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.

15.1.8 Timer Expiration Messages

For details about timer expiration messages, see section 16.4.

15.1.9 Partition Reference Time Enlightenment

The partition reference time enlightenment presents a reference time source to 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 fall-back 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.

15.2 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)

15.2.1 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.

15.3 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

15.3.1 Synthetic Timer Configuration Register

63:20	19:16	15:4	3	2	1	0
RsvdZ	SINTx	RsvdZ	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
15:3	RsvdZ (value should be set to zero)	Read/write

Bits	Description	Attributes
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.

15.3.2 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.

15.4 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.

15.4.1 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.

15.4.2 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;
```

15.4.3 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.

15.4.3.1 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.

15.4.3.2 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. The valid values of this field range from 0 to 0xFFFFFFFE.

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 (e.g. virtual PM timer).

15.4.3.3 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 virtual PM timer).The recommended code for computing the partition reference time using this enlightenment is shown below:

```
Repeat:
      Sequence = Stats->ReferenceTscSequence;
      Tsc = RDTSC();
      Scale = Stats->ReferenceTscScale;
      Offset = Stats->ReferenceTscOffset;
      ReferenceTime = (Tsc * Scale) >> 64 + Offset;

      If (Stats->Sequence == Sequence)
      {
      return ReferenceTime;
      }

      If (Stats->Sequence != 0x0)
      {
      goto Repeat;  // Fall back to a different method to get
reference time
      }
```

16 Message Formats

16.1 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 14The 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

16.2 Message Data Types

Intercept messages are delivered by the SynIC. For a description of SynIC messages, including the message header layout, see chapter 14**.**

16.2.1 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 14.8.4.

16.2.2 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.

16.2.3 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:9;
} 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 10.3.4.

16.2.4 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.

16.2.5 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.

16.2.6 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.

16.3 Memory Access Messages

Memory access messages are sent in response to certain memory accesses performed by a virtual processor or performed on its behalf by the hypervisor. Both memory access messages (unmapped GPA and GPA access violations) share the same format.

Message Header	0	MessageType (4 bytes)			Rsvd (3 bytes)	PayloadSize (1 byte)
	8	SourcePartition (8 bytes)				
Memory Access Payload	16	VpIndex (4 bytes)		InstLen (1 byte)	Access Type (1 byte)	ExecutionState (2 bytes)
	24	CsSegment[0] (8 bytes)				
	32	CsSegment[1] (8 bytes)				
	40	Rip (8 bytes)				
	48	Rflags (8 bytes)				
	56	Rsvd (2 bytes)	Flags (1	InstByteCount	CacheType (4 bytes)	

			byte)	(1 byte)	
	64	Gva (8 bytes)			
	72	Gpa (8 bytes)			
	80	InstructionBytes[0]			
	88	InstructionBytes[1]			
	96	DsSegment[0] (8 bytes)			
	104	DsSegment[1] (8 bytes)			
	112	SsSegment[0] (8 bytes)			
	120	SsSegment[1] (8 bytes)			
	128	Rax (8 bytes)			
	⋮	⋮			
	248	R15 (8 bytes)			

Flags is of type HV_X64_MEMORY_ACCESS_FLAGS and provides information about the intercept.

InstByteCount indicates how many instruction bytes have been provided in the *InstructionBytes* fields. Valid values are in the range 0 through 16.

CacheType is of type HV_CACHE_TYPE (see section 12.3.3). When *GvaValid* is set, this field is the cache type for the page causing the access intercept obtained from the guest's page table. When *GvaValid* is clear, the intercept was caused by an access to guest pages table pages and this field will be HvCacheTypeX64WriteBack.

Gva indicates the guest virtual address of the access that was intercepted. This field is valid if the bit *GvaValid* is set in *Flags*. GvaValid will not be set if the memory intercept was caused by an access to guest page table pages.

Gpa indicates the guest physical address of the access that was intercepted. If the intercepted instruction was attempting to perform a multi-byte memory access that crossed page boundaries, this value will indicate the lowest address that resulted in an intercept.

InstructionBytes includes up to 16 bytes from the instruction stream starting at the current CS:RIP. The number of valid bytes is specified by *InstByteCount*. If the access was due to the delivery of an interruption (as indicated by the *InterruptionPending* bit in the *ExecutionState* field) then the message will not include any instruction bytes. In other cases, the hypervisor may fetch only up to the next page boundary or up to the end of the code segment resulting in an instruction byte count that is less than 16.

DsSegment provides the selector, base, limit, and flags for the current data segment (DS).

SsSegment provides the selector, base, limit, and flags for the current stack segment (SS).

Rax...R15 provides the current values of the virtual processor's sixteen 64-bit general purpose registers.

16.3.1 Unmapped GPA Message

An “unmapped GPA” message is delivered by the hypervisor when a virtual processor (or the hypervisor on its behalf) accesses a GPA page for which no mapping has been indicated.

16.3.2 GPA Access Violation Message

A “GPA access violation” message is delivered by the hypervisor when a virtual processor (or the hypervisor on its behalf) accesses a GPA page and the access type is not allowed by the current mapping.

16.4 Timer Messages

16.4.1 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	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.

16.5 Processor Event Messages

Processor event messages are sent in response to certain actions performed by a virtual processor.

16.5.1 CPUID Intercept Message

A “CPUID intercept” message is delivered by the hypervisor when a virtual processor executes a CPUID instruction and the parent has installed an intercept on such instructions.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	SourcePartition (8 bytes)		

CPUID Intercept Payload	16	VpIndex (4 bytes)	InstLen (1 byte)	Access Type (1 byte)	ExecutionState (2 bytes)
	24	CsSegment[0] (8 bytes)			
	32	CsSegment[1] (8 bytes)			
	40	Rip (8 bytes)			
	48	Rflags (8 bytes)			
	56	Rax (8 bytes)			
	64	Rcx (8 bytes)			
	72	Rdx (8 bytes)			
	80	Rbx (8 bytes)			
	88	DefaultResultRax (8 bytes)			
	96	DefaultResultRcx (8 bytes)			
	104	DefaultResultRdx (8 bytes)			
	112	DefaultResultRbx (8 bytes)			

Rax-Rbx provides the values in the corresponding registers when the CPUID instruction is executed.

DefaultResultRax-DefaultResultRbx provides the default return values that the hypervisor would have returned in response to the CPUID instruction if the intercept had not been requested. The intercept handler is free to override these default values.

16.5.2 MSR Intercept Message

A “MSR intercept” message is delivered by the hypervisor when a virtual processor executes a RDMSR or WRMSR instruction and the parent has installed an intercept on the specified MSR.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)		PayloadSize (1 byte)
	8	SourcePartition (8 bytes)			
MSR Intercept Payload	16	VpIndex (4 bytes)	InstLen (1 byte)	Access Type (1 byte)	ExecutionState (2 bytes)
	24	CsSegment[0] (8 bytes)			
	32	CsSegment[1] (8 bytes)			
	40	Rip (8 bytes)			
	48	Rflags (8 bytes)			

	56	MsrNumber (4 bytes)	Rsvd (4 bytes)
	64	Rdx (8 bytes)	
	72	Rax (8 bytes)	

MsrNumber provides the value in the RCX register that indicates the index of the MSR being accessed.

Rdx provides the value in the RDX register. For writes, it represents the top half of the value being written to the MSR.

Rax provides the value in the RAX register. For writes, it represents the bottom half of the value being written to the MSR.

16.5.3 I/O Port Intercept Message

An “I/O Port Intercept” message is delivered by the hypervisor when a virtual processor executes an IN, OUT, INS, or OUTS instruction and the parent has installed an intercept on one of the accessed I/O ports. A multi-byte I/O port access is treated as though it accesses multiple consecutive ports. If intercepts have been installed for any of the ports within the range, an intercept is generated.

Message Header	0	MessageType (4 bytes)		Rsvd (3 bytes)		PayloadSize (1 byte)
	8	SourcePartition (8 bytes)				
IO Port Intercept Payload	16	VpIndex (4 bytes)		InstLen (1 byte)	Access Type (1 byte)	ExecutionState (2 bytes)
	24	CsSegment[0] (8 bytes)				
	32	CsSegment[1] (8 bytes)				
	40	Rip (8 bytes)				
	48	Rflags (8 bytes)				
	56	PortNum (2 bytes)	Access Info (1 byte)	InstByteCount (1 byte)	Rsvd (4 bytes)	
	64	Rax (8 bytes)				
	72	InstructionBytes[0] (8 bytes)				
	80	InstructionBytes[1] (8 bytes)				
	88	DsSegment[0] (8 bytes)				
	96	DsSegment[1] (8 bytes)				
	104	EsSegment[0] (8 bytes)				
	112	EsSegment[1] (8 bytes)				

	120	Rcx (8 bytes)
	128	Rsi (8 bytes)
	136	Rdi (8 bytes)

PortNum provides the number of the I/O port being accessed.

AccessInfo is of type HV_X64_IO_PORT_ACCESS_INFO and indicates the size of the access, whether the intercepted instruction was a string operation, and whether a “rep” prefix was specified.

InstByteCount indicates how many instruction bytes have been provided in the InstructionBytes fields. Valid values are in the range 0 through 16. This field is valid only for string operations. It is set to zero and should be ignored for simple (non-string) I/O port accesses.

Rax provides the value of the Rax register. For OUT instructions, it represents the value written to the I/O port. For IN instructions, it can be used by an intercept handler to insert the input value into AL, AX, or EAX and compute the final value of RAX.

The remaining fields (offsets 72 and beyond) are applicable only for string operations. They are set to zero and should be ignored for simple (non-string) I/O port accesses.

InstructionBytes includes up to 16 bytes from the instruction stream starting at the current CS:RIP. The number of valid bytes is specified by *InstByteCount*. The hypervisor may only fetch up to the next page boundary resulting in an instruction byte count that is less than 16.

DsSegment provides the selector, base, limit, and flags for the current data segment (DS).

EsSegment provides the selector, base, limit, and flags for the current extended segment (ES).

Rcx, Rsi, and *Rdi* provide the values of these three registers.

16.5.4 Exception Intercept Message

An “exception intercept” message is delivered by the hypervisor when a virtual processor generates an exception.

Message Header	0	MessageType (4 bytes)		Rsvd (3 bytes)		PayloadSize (1 byte)
	8	SourcePartition (8 bytes)				
Exception Intercept Payload	16	VpIndex (4 bytes)		InstLen = 0 (1 byte)	Access Type (1 byte)	ExecutionState (2 bytes)
	24	CsSegment[0] (8 bytes)				
	32	CsSegment[1] (8 bytes)				
	40	Rip (8 bytes)				
	48	Rflags (8 bytes)				
	56	Exception Vector	Exception	InstrByte	ErrorCode (4 bytes)	

		(2 bytes)	Flags (1 byte)	Count (1 byte)	
	64	ExceptionParameter (8 bytes)			
	72	Rsvd (8 bytes)			
	80	InstructionBytes[0] (8 bytes)			
	88	InstructionBytes[1] (8 bytes)			
	96	DsSegment[0] (8 bytes)			
	104	DsSegment[1] (8 bytes)			
	112	SsSegment[0] (8 bytes)			
	120	SsSegment[1] (8 bytes)			
	128	Rax (8 bytes)			
	⋮	⋮			
	248	R15 (8 bytes)			

ExceptionVector provides the vector number of the exception that was generated.

ExceptionFlags is of type HV_X64_EXCEPTION_INFO and provides information about the exception.

InstByteCount indicates how many instruction bytes have been provided in the InstructionBytes fields. Valid values are in the range 0 through 16.

ErrorCode provides the error code that would have been pushed as part of the exception frame. This field is valid only if the exception flags indicate that an error code is present.

ExceptionParameter provides additional information whose meaning is specific to the exception type. It is only defined for page faults (vector 0x0E) and breakpoint exceptions (vector 0x01). In the case of page faults, the virtual address that caused the fault is provided (that is, the value that would normally be placed in CR2 by the processor). In the case of breakpoint exceptions, the value that would have been placed in DR6 is provided. This field is reserved for all other exception type.

InstructionBytes includes up to 16 bytes from the instruction stream starting at the current CS:RIP. The number of valid bytes is specified by *InstByteCount*. If the access was due to the delivery of an interruption (as indicated by the *InterruptionPending* bit in the *ExecutionState* field), the message will not include any instruction bytes. In other cases, the hypervisor may only fetch up to the next page boundary or up to the end of the code segment, resulting in an instruction byte count that is less than 16.

DsSegment provides the selector, base, limit, and flags for the current data segment (DS).

SsSegment provides the selector, base, limit, and flags for the current stack segment (SS).

Rax...R15 provides the current values of the virtual processor’s sixteen 64-bit general purpose registers.

16.5.5 APIC EOI Message

An APIC EOI message is delivered by the hypervisor after a virtual processor executes an instruction that writes to the APIC's memory-mapped EOI register or APIC EOI MSR, indicating the "end of interrupt". This message is generated only for level-triggered fixed or external interrupts.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	SourcePartition (8 bytes)		
APIC EOI	16	VpIndex (4 bytes)	InterruptVector (4 bytes)	

VpIndex provides the index of the virtual processor.

InterruptVector is the vector of the interrupt that was just EOIed.

16.5.6 FERR Asserted Message

A FERR Asserted (legacy floating point support) message is delivered by the hypervisor after a virtual processor executes an instruction that generates an unmasked floating point error when CR0.NE is zero and the IGNNE# bit (in HvX64RegisterFpControlStatus) is zero.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	SourcePartition (8 bytes)		
Legacy FP	16	VpIndex (4 bytes)	Rsvd (4 bytes)	

VpIndex provides the index of the virtual processor.

16.5.7 Invalid VP Register Value Message

When a virtual processor's registers are modified, it is possible to specify an illegal register value or combination of register values. In this case, a "bad VP register value" message is generated when the virtual processor resumes execution.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	SourcePartition (8 bytes)		
Invalid Register Payload	16	VpIndex (4 bytes)	Rsvd (4 bytes)	

VpIndex provides the index of the virtual processor.

16.5.8 Unrecoverable Exception Message

An “unrecoverable exception” message is delivered by the hypervisor when a virtual processor generates an exception that cannot be delivered (for example, a triple fault on x64).

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)		PayloadSize (1 byte)
	8	SourcePartition (8 bytes)			
Unrecoverable Exception Payload	16	VpIndex (4 bytes)	InstLen = 0 (1 byte)	AccessType (1 byte)	ExecutionState (2 bytes)
	24	CsSegment[0] (8 bytes)			
	32	CsSegment[1] (8 bytes)			
	40	Rip (8 bytes)			
	48	Rflags (8 bytes)			

16.5.9 Unsupported Feature Message

This message is generated when a virtual processor accesses a feature of the architecture that is not properly virtualized by the hypervisor.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	SourcePartition (8 bytes)		
Unsupported Feature Payload	16	VpIndex (4 bytes)	FeatureCode (4 bytes)	
	24	FeatureParameter (8 bytes)		

VpIndex provides the index of the virtual processor.

FeatureCode provides a code that indicates which feature was accessed. Codes are defined in section 9.2.4.

FeatureParameter provides additional information whose meaning is specific to the *FeatureCode*. In the case of an unsupported intercept, the intercept number is supplied. In the case of an unsupported TSS in a task switch request, the TSS type is supplied.

16.5.10 Event Log Buffers Ready Message

This message is sent to the root partition when either:

the number of pending ready event log buffers reaches the threshold established by the guest using the HvInitializeEventLogBufferGroup hypercall, or

the guest has requested that all pending complete and in-use event log buffers be flushed.

Event Logging support is described in detail in chapter 19.

Message Header	0	MessageType (4 bytes)	Rsvd (3 bytes)	PayloadSize (1 byte)
	8	Rsvd (8 bytes)		
Buffer Complete Payload	16	EventLogType (4 bytes)	BufferIndex (4 bytes)	

EventLogType provides the type of the buffer that has been placed into a “complete” state.

BufferIndex is the index of the first buffer in a ready buffer list. The *NextBufferIndex* field of the event log buffer header (in the message payload) will contain the index of the next buffer in the list. The end of the list is indicated by a buffer index matching the value HV_EVENTLOG_BUFFER_INDEX_NONE. Buffers appear in the list in the order in which they were placed into the “complete” state. If the event log buffer type uses buffers of the local buffer class, then the list contains only buffers that were completed by this logical processor.

17 Partition Save and Restore

The hypervisor provides a means to store certain state of a partition and later restore this state. This support allows a caller to resume execution of the partition at a later time—potentially on a different piece of hardware.

17.1 Overview

17.1.1 Saved State

The hypervisor allows a parent partition to save the state of a child partition. The *saved state* may be migrated (restored) on another system or check pointed for a subsequent restart.

All virtual processors must be explicitly suspended before a partition can be saved.

17.1.2 Summary State

In addition to *saved state*, the hypervisor allows a caller to retrieve *summary state* data for the partition or system. Using this state, the caller can determine whether a partition's potential complete saved state is specifically compatible with respect to the underlying target hardware and hypervisor implementation.

Summary state consists of a minimal amount of information relating specifically to hardware and hypervisor compatibility and does not include most partition-specific context information. Unlike saved state, summary state can be retrieved without the necessity of suspending all of the partition's virtual processors in order to obtain it.

The HvSavePartitionState and HvRestorePartitionState hypercalls are used to retrieve and subsequently test compatibility using the summary state. The summary state can be related to the system and, optionally, to a particular partition on the system. Absence of a partition ID indicates that system information alone is to be evaluated. Presence of a partition ID indicates that both system and partition-related information (for that particular partition) is to be evaluated.

17.1.3 Saved State Compatibility and Versioning

The partition and virtual processor state saved from one version of the hypervisor is not guaranteed to be compatible with future versions of the hypervisor. Attempts will be made to minimize version incompatibilities.

Some differences between hardware platforms also introduce saved state incompatibilities, so it may not be possible to save a partition's state on one physical system and restore it on a second physical system with sufficiently different hardware.

The list of potential incompatibilities is not architecturally defined, but the following list provides some potential examples of hardware incompatibilities that may result in an inability to save and restore:

- Virtual processors that are executing real-mode code on CPUs developed by Intel may not be restorable on CPUs developed by AMD.
- Partitions that are executing on logical processors that support various guest-visible features (for example, 3DNow!, SSE3, and so on.) may not be restorable on a system whose logical processors do not support these features.

17.1.4 State That Is Not Saved by the Hypervisor

In general, state that is explicitly managed and set by the parent partition is not saved by the hypervisor. It is expected that code running in the parent partition will save and restore this information.

Information that is *not* saved by the hypervisor includes:

- Partition creation flags
- Partition properties
- Scheduling policy settings and affinities
- Partition ID or parent ID
- GPA mappings and access rights
- Contents of pages
- Intercepts
- Memory pool balance
- Ports and connections
- Partition state
- Default CPUID return values
- Partition statistics
- Monitored notification context

17.1.5 State That Is Saved by the Hypervisor

The specific state saved may differ from one hypervisor implementation to the next. The format of the saved state is purposely not defined.

In general, state that is not explicitly managed and set by the parent partition and is potentially changed by a guest running within the child partition is saved.

Partition state that is saved includes:

- Partition version information
- Logical processor information (sufficient to determine compatibility for restoration)
- Partition-wide MSR values (for example, hypercall base address)
- Locations of overlay pages
- Reference counter

Virtual processor state that is saved includes:

- All virtual processor registers
- Undelivered (queued) messages
- APIC and SynIC state
- SIEF and SIM page contents
- Synthetic and periodic timers
- Periodic and PM timer assists

17.1.6 Partition State Streams

A partition’s state is saved as a stream of data. The format of the stream is not defined and may change between different versions of the hypervisor or different platforms supported by a single hypervisor.

Failure to save and restore the complete save state (see section 17.1.1) of a partition may result in a partition that functions incorrectly or not at all. In particular, an “invalid state” error may (but is not guaranteed to) be generated when its virtual processors are unsuspended.

Changes to the partition or its virtual processors will necessarily change the state. Care should be taken to avoid modifying a partition or its virtual processors in the middle of the save or restore process. Failure to follow this rule may result in a partition that functions incorrectly after restoration.

Summary state (see section 17.1.2) is intended solely for testing a partition's save-restore compatibility and cannot be used to affect any partition-specific context.

17.1.7 Recommended Save Process

It is recommended that a parent partition use the following procedure to save a partition.

- Suspend all of the virtual processors within a partition by calling HvSetVpRegisters and modifying the "explicit suspend" register (see chapter 10). Note that it is possible for virtual processors to already be suspended due to intercepts. It is recommended that all outstanding intercepts be addressed prior to proceeding.
- Drain and process all outstanding messages on ports with a connection from the child partition being saved (see chapter 17). If the parent generates messages to be sent back to the child, it should be prepared to save any messages that it was unable to successfully post. Suspension of the child's virtual processors precludes the child's ability to receive messages and could lead to a lack of message buffer availability on any of the child's ports. The parent may elect to include the drained messages with its saved context for execution after the child partition's restoration (and subsequent resumption) with its new parent.
- Call HvSavePartitionState (see section 17.3.1) repeatedly until the entire stream of state data has been retrieved.

17.1.8 Recommended Restore Process

It is recommended that a parent partition use the following procedure to restore a partition.

- Call HvRestorePartitionState supplying the saved or summary data and appropriate flags to determine whether the partition state can be restored (see section 17.3.2).
- Create a new partition (see section 5.6.1) and populate it with resources, such as GPA mappings (see chapters 7 and 8) and so on.
- Call HvInitializePartition to initialize the new partition (see section 5.6.2).
- Create new virtual processors. The count and IDs of the virtual processors should match those of the partition that was previously saved. (see chapter 10).
- Recreate all child ports and reestablish all connections with the new parent (see chapter 14).
- Call HvRestorePartitionState repeatedly, supplying the saved state stream until it indicates that the partition's state has been completely restored (see section 17.3.2).
- Resume execution by clearing the "explicit suspend" state of each of the virtual processors (see chapter 10).

17.2 Partition Save and Restore Data Types

17.2.1 Partition Save and Restore State

The calls HvSavePartitionState and HvRestorePartitionState are expected to be called several times in succession until the complete state has either been returned (save) or restored (restore). When the hypercall return value indicates successful completion, a separate status representing the *overall state* of the save and restore is returned with each of the calls. The *overall state* provides updated feedback concerning the save or restore process and is used to indicate completion, interim success (incomplete) or an error. The errors indicate that continuation is not possible and why. Note that while a hypercall may have succeeded (as indicated by HV_STATUS_SUCCESS in the hypercall return value), the *overall state* may return an error. The *overall state* should not be examined in the event of a hypercall failure, in accordance with the hypercall status rules described in section 4.11.

```
typedef enum
{
      HvStateComplete   = 0,
      HvStateIncomplete = 1,
      HvStateRestorable = 2,
      HvStateCorruptData      = 3,
      HvStateUnsupportedVersion     = 4,
      HvStateProcessorFeatureMismatch     = 5,
      HvStateHardwareFeatureMismatch      = 6,
      HvStateProcessorCountMismatch = 7,
      HvStateProcessorFlagsMismatch = 8,
      HvStateProcessorIndexMismatch = 9,
      HvStateProcessorInsufficientMemory                = 10,
    HvStateIncompatibleProcessor                      = 11,
    HvStateProcessorFeatureSse3Mismatch               = 12,
    HvStateProcessorFeatureLahfSahfMismatch           = 13,
    HvStateProcessorFeaturSsse3eMismatch              = 14,
    HvStateProcessorFeatureSse41Mismatch              = 15,
    HvStateProcessorFeatureSse42Mismatch              = 16,
    HvStateProcessorFeatureSse4aMismatch              = 17,
    HvStateProcessorFeatureSse5Mismatch               = 18,
    HvStateProcessorFeaturePopcntMismatch             = 19,
    HvStateProcessorFeatureCmpxchg16bMismatch         = 20,
    HvStateProcessorFeatureAltmovcr8Mismatch          = 21,
    HvStateProcessorFeatureLzcntMismatch              = 22,
    HvStateProcessorFeatureMisalignedSseMismatch      = 23,
    HvStateProcessorFeatureMmxExtMismatch             = 24,
    HvStateProcessorFeature3DNowMismatch              = 25,
    HvStateProcessorFeatureExtended3DNowMismatch      = 26,
    HvStateProcessorFeaturePage1GBMismatch            = 27,
    HvStateProcessorCacheLineFlushSizeMismatch        = 28,
    HvStateProcessorFeatureXsaveMismatch              = 29,
    HvStateProcessorFeatureXsaveoptMismatch           = 30,
    HvStateProcessorFeatureXsaveLegacySseMismatch     = 31,
    HvStateProcessorFeatureXsaveAvxMismatch           = 32,
    HvStateProcessorFeatureXsaveUnknownFeatureMismatch = 33,
    HvStateProcessorXsaveSaveAreaMismatch             = 34
} HV_SAVE_RESTORE_STATE_RESULT;

typedef HV_SAVE_RESTORE_STATE_RESULT *PHV_SAVE_RESTORE_STATE_RESULT;
```

The following table describes the meaning of these values. Applicability to save and restore is shown.

Value	Save	Restore	Meaning
HvStateComplete	✔	✔	For HvSavePartitionState, the complete state of the partition has been returned and no more state data is available. For HvRestorePartitionState, the complete state of the partition has been restored and no more state data is expected.
HvStateIncomplete	✔	✔	The partition save or restore process has not completed and additional calls (HvSavePartitionState or HvRestorePartitionState) are required. At this point in the save or restore process no error condition has thus far been detected.
HvStateRestorable		✔	The partition's potential complete saved state is specifically compatible with respect to the underlying target hardware and hypervisor implementation.
HvStateCorruptData		✔	The partition state appears to be corrupted.
HvStateUnsupportedVersion		✔	The hypervisor implementation is not compatible with the version of the saved state.
HvStateProcessorFeatureMismatch		✔	The features supported by the logical processor(s) are incompatible with the features of the logical processor(s) in use when the state was saved.
HvStateHardwareFeatureMismatch		✔	The features supported by the physical hardware are incompatible with the features of the physical hardware in use when the state was saved.
HvStateProcessorCountMismatch		✔	The number of virtual processors associated with the restore partition is different from the number of virtual processors present when the partition's state was saved.
HvStateProcessorFlagsMismatch		✔	The option flags associated with a virtual processor in the restore partition mismatches those associated with the equivalent virtual processor when the state was saved.
HvStateProcessorIndexMismatch		✔	The indexes of the virtual processors in the restore partition differ from those associated with the virtual processors when the state was saved.

17.2.2 Save and Restore Partition State Flags

The following flags are used with the HvSavePartitionState and HvRestorePartitionState hypercalls.

```
typedef UINT32 HV_SAVE_RESTORE_STATE_FLAGS;

#define HV_SAVE_RESTORE_STATE_START 0x00000001
#define HV_SAVE_RESTORE_STATE_SUMMARY     0x00000002
```

The flag HV_SAVE_RESTORE_STATE_START is required to begin a save or restore process and should be specified with the first invocation of HvSavePartitionState or HvRestorePartitionState. Subsequent invocations should not specify this flag unless it is prematurely terminating a save or restore process. In this case it is used to ensure that all intervening save or restore context maintained by the hypervisor is released.

The flag HV_SAVE_RESTORE_STATE_SUMMARY indicates that the save data is summary data (see section 17.1.2) and not complete partition save data.

The HV_SAVE_RESTORE_STATE_START and HV_SAVE_RESTORE_STATE_SUMMARY flags may both be present.

17.3 Partition Save and Restore Interfaces

17.3.1 HvSavePartitionState

The HvSavePartitionState hypercall saves part of the state of a partition. It may also be used to obtain summary state data for the system or a partition.

Wrapper Interface

```
HV_STATUS
HvSavePartitionState(
      __in  HV_PARTITION_ID         PartitionId,
      __in  HV_SAVE_RESTORE_STATE_FLAGS   Flags,
      __out PHV_SAVE_RESTORE_STATE_RESULT SaveState,
      __out PUINT32           SaveDataCount,
      __out bcount_part(4080, *SaveData),
            PVOID                         SaveData
      );
```

Native Interface

HvSavePartitionState		
	Call Code = 0x005E	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	Flags (4 bytes)	Padding (4 bytes)
⬅ Output Parameters		
0	SaveState (4 bytes)	SaveDataCount (4 bytes)
8	SaveData (up to 4080 bytes)	

Description

For information regarding summary state data for the system or a partition, see section 17.1.2. For a description of the recommended save process, see section 17.1.7.

Save data is returned in a sequence-sensitive stream. The caller begins the save process by specifying the HV_SAVE_RESTORE_STATE_START flag. Subsequent calls should not specify this flag. The process should progress as long as the call to HvSavePartitionState returns success and *SaveState* indicates HvStateIncomplete. The process ends either when an error status is returned or success is returned and *SaveState* indicates HvStateComplete.

Input Parameters

PartitionId specifies the partition ID of the partition that is being saved, or zero if system summary state data is being retrieved.

Flags specifies what data are to be saved.

Output Parameters

SaveDataCount indicates how many bytes of state were saved (retrieved).

SaveData contains some or all of the requested save state, the size of which is returned in *SaveDataCount.* At most 4080 bytes can be returned and, as with all hypercalls, the output parameters can never cross a page boundary. A size of zero is legal and does not have any significance.

SaveState contains the status feedback of the overall save procedure.

Restrictions

The partition specified by *PartitionId* must be in the “active” state.

The caller must be the parent of the partition specified by *PartitionId.*

All virtual processors must be in an explicit suspended state except when saving summary information.

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_OPERATION_DENIED	The specified partition has one or more virtual processors that are not explicitly suspended. (This does not apply when saving summary information.)
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.
HV_STATUS_INVALID_PARAMETER	The flags parameter specifies an unsupported option.
HV_STATUS_INVALID_SAVE_RESTORE_STATE	HvSavePartitionState has not been called with HV_SAVE_RESTORE_STATE_START to initialize the save process.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the caller is insufficient to perform the operation.

17.3.2 HvRestorePartitionState

The HvRestorePartitionState hypercall restores partition state that was previously saved. It may also be used to submit summary state data for a system or partition for compatibility assessment purposes.

Wrapper Interface

```
HV_STATUS
HvRestorePartitionState(
      __in    HV_PARTITION_ID        PartitionId,
      __in    HV_SAVE_RESTORE_STATE_FLAGS  Flags,
      __inout PUINT32                      RestoreDataCount,
      __in    bcount(RestoreDataCount),
              PCVOID                       RestoreData,
      __out   HV_SAVE_RESTORE_STATE_RESULT RestoreState
      );
```

Native Interface

HvRestorePartitionState		
	Call Code = 0x005F	
➡ Input Parameters		
0	PartitionId (8 bytes)	
8	Flags (4 bytes)	RestoreDataCount (4 bytes)
16	RestoreData (up to 4080 bytes)	
⬅ Output Parameters		
0	RestoreState (4 bytes)	RestoreDataConsumed (4 bytes)

Description

For information regarding summary state data for a system or partition, see section 17.1.2.

The caller should make this call repeatedly until the entire save stream has been presented to the hypervisor. For the recommended restore process, see section 17.1.8.

Input Parameters

PartitionId specifies the partition ID of the partition that is being restored, or zero if system summary state data is being presented for compatibility assessment purposes.

Flags specifies which data are to be restored.

RestoreDataCount supplies the number of valid bytes provided in *RestoreData.*

RestoreData provides some or all of the (previously saved) state that is to be restored. At most 4080 bytes can be provided and, as with all hypercalls, the input parameters can never cross a page boundary.

Output Parameters

RestoreState receives the status feedback of the overall restore procedure.

RestoreDataConsumed returns the number of bytes of *RestoreData* that has been processed by the hypervisor. This value may range from zero to the value provided in *RestoreDataCount.*

Restrictions

The caller must possess the *CreatePartitions* privilege.

The partition specified by *PartitionId* must be in the “active” state.

The caller must be the parent of the partition specified by *PartitionId.*

All virtual processors must be in an explicit suspend state.

Return Values

Status code	Error condition

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the parent of the specified partition. The caller's partition privilege flag *CreatePartitions* is cleared.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid.
HV_STATUS_OPERATION_DENIED	The specified partition has one or more virtual processors that are not explicitly suspended.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.
HV_STATUS_INVALID_PARAMETER	The flags parameter specifies an unsupported option.
HV_STATUS_INVALID_SAVE_RESTORE_STATE	HvRestorePartitionState has not been called with HV_SAVE_RESTORE_STATE_START to initialize the restoration process.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the caller is insufficient to perform the operation.

18 Scheduler

18.1 Scheduling Concepts

The hypervisor schedules virtual processors to run on logical processors. The interfaces in this section allow a parent partition to set the scheduling policy for its child partitions.

The hypervisor scheduler makes scheduling decisions based on these policy settings. Because these decisions are made at discreet times, not continuously, the policies are guaranteed only over a sufficiently long period of time. This time is purposely undefined but will typically be on the order of hundredths of a second.

18.2 Scheduling Policy Settings

Various schedule policy settings can be obtained using HvGetPartitionProperty (see section 5.6.5) and adjusted using the HvSetPartitionProperty (see section 5.6.6) hypercall interfaces. These include:

- CPU Reserve
- CPU Cap
- CPU Weight

Each of these is described in more detail in the following sections.

18.2.1 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 18.2.2).

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. Reserves are not guaranteed if the number of virtual processors in a partition exceeds the number of logical processors that are available for scheduling.

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.

18.2.2 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 18.2.1).

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.

18.2.3 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.

18.3 Other Scheduling Considerations

18.3.1 Hyperthreading

Multiple virtual processors can optionally be grouped together and scheduled onto hyperthreads within a single physical processor core. In effect, these virtual processors then act like virtual hyperthreads. When virtual processors are grouped as such, the hypervisor tries to schedule them concurrently on the same physical processor core. This scheduling behavior potentially improves performance and reduces information leakage across partition boundaries.

18.3.2 NUMA and Affinity

When a virtual processor is run-able, the hypervisor's scheduler assigns it to a logical processor. The placement is determined based on a variety of factors including workload, reservations, and NUMA topology. In general, the scheduler will attempt to keep a virtual processor scheduled on a logical processor that is topologically closest to the memory being accessed by the virtual processor, in effect minimizing memory access times.

The scheduler also attempts to create as much temporal affinity as possible. That is, it will prefer to run a virtual processor on the same logical processor each time it is scheduled. If the logical processor is oversubscribed, the scheduler may move it to another logical processor.

18.3.3 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. Through this hypercall, a guest notifies the hypervisor of a long spinlock acquisition. This allows the hypervisor to make better scheduling decisions.

18.4 Scheduler Data Types

The following data types support the scheduler interfaces.

```
typedef UINT64 HV_INPUT_NOTIFY_LONG_SPINWAIT ;
typedef HV_INPUT_NOTIFY_LONG_SPINWAIT,
*PHV_INPUT_NOTIFY_LONG_SPINWAIT;
```

18.5 Scheduler Interfaces

18.5.1 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.

19 Event Logging

19.1 Overview

The hypervisor provides a general mechanism for recording events within logs and exposing the resulting data, via buffers, to the root partition. Through this mechanism, the root partition can selectively collect various types of events, such as:

- Security logs (for example, policy and rights modifications)
- Diagnostic logs (for example, hardware or software failures)
- Performance analysis information (for example, call traces and profiler data)
- Self-test results

While the hypervisor defines the common framework to be used for buffers and events within buffers, it leaves the definition of the event data itself to the particular application.

19.1.1 Event Log Buffers

Event log buffers are composed of between 1 and 512 pages of memory, making them between 4K and 2MB in size. Buffers are created individually, and the size and limit are determined from the event log buffer group's geometry (geometry is described in the following section). The pages that constitute each buffer are allocated from the partition's pool.

When a buffer is created, its pages are allocated and initialized. It remains inaccessible to the guest however, and must therefore be mapped prior to access. Mapping the buffer creates read-only overlay pages in the guest's address space and returns the GPAs of the pages that constitute the buffer. The buffer remains read-only accessible until it is unmapped. Deletion of the buffer returns the buffer's pages to the partition's pool from whence they were allocated.

19.1.2 Event Log Buffer Groups

Associated with each event log type is an *event log buffer group* that has a fixed set of characteristics. These include the buffer *geometry* and the *completed buffer threshold*. The geometry is defined as the *maximum number of buffers* and the *number of pages per buffer*. All buffers created as part of the group are the same size (that is, they are composed of the same number of pages). A maximum number of 512 buffers may be in the group at any one time. Depending upon the size of the buffers, the hypervisor implementation may not support the creation of the maximum number of buffers.

Prior to use, the event log buffer group must be initialized, at which time the group's characteristics are established. Initialization of the group may require memory to manage the necessary infrastructure to support the event log type. This will be allocated from the partition's pool. When the group is no longer needed, event logging should be disabled, all buffers should be unmapped and deleted, and the group subsequently destroyed.

19.1.3 Local and Global Buffer Classes

The hypervisor supports two event log buffer classes:

Local buffers are distributed across and associated with specific logical processors and can be filled without acquiring costly cross-processor locks. The local buffer class is used with performance-critical applications. When deriving the event log buffer group geometry, the guest should consider the number of logical processors and the NUMA topology.

Global buffers are shared among all logical processors, requiring their access to be synchronized using global locks. The global buffer class is used for applications that are not performance critical or where the frequency of event generation is low.

The guest should allocate multiple event log buffers for the following reasons:

For local buffers, at least one buffer should be available as the “in use” (currently active) buffer for each active logical processor.

When there is insufficient space available for a new event log entry in the active buffer, the hypervisor will “complete” it. Buffers may also be completed prematurely by the guest requesting the flushing active buffers. If, at the time that an event is ready to be logged, no “in use” buffer is present, then a “free” buffer will be allocated and become the “in use” buffer. If local buffers are being used, only buffers created for that specific logical processor will be used. To avoid interruption in the logging of events, it is important for there to be sufficient free buffers available.

19.1.4 Event Log Buffer Indices

Buffers are assigned an *event log buffer index* beginning at zero and ending at the maximum number of buffers (established at group initialization) minus one. The index is unique within the event log type and is used by both the hypervisor and the guest to identify each individual buffer in a consistent manner. The index associated with each buffer is assigned by the guest when the buffer is created. An attempt to create a buffer with an index that exceeds the maximum or that matches one assigned to an existing buffer will result in an error.

19.1.5 Event Log Buffer States

When a buffer is created, it is part of the event log buffer group associated with an event log type. It is initialized in an internal “standby” state. While in this state, the buffer is neither accessible from the guest nor usable by the hypervisor. Mapping the buffer reveals the location of its constituent pages and makes the buffer available to the hypervisor for recording events. Un-mapping the buffer returns it to the internal “standby” state.

While the event log type is enabled, all related buffers of the group have a guest-visible state described as follows:

HvEventLogBufferStateFree – The buffer is currently available to be allocated by the hypervisor as an “in use” buffer. With the exception of the header’s state field, the guest should not read the buffer’s contents. A buffer enters the free state when it is either mapped or released by the guest. It remains in this state until it is either unmapped by the guest (hence, taken out of service) or allocated by the hypervisor as an “in use” buffer. Note that the hypervisor may change the state of this buffer at any time.
HvEventLogBufferStateInUse – The buffer is currently active and accepting event messages. Buffers of this type are being written by the hypervisor and may be accessed by the guest in a limited fashion (see section 19.1.12 for details). Buffers leave the “in use” state when they are full or when the guest requests that buffers be flushed. Note that the buffer can become full at any time and the hypervisor could therefore change the state at any time.
HvEventLogBufferStateComplete – The buffer is no longer active and contains valid data. Buffers of this type are no longer being written to by the hypervisor and may be read by the guest. Buffers in the “complete” state are on a list waiting for the guest to be notified of their completion. Buffers leave the HvEventLogBufferStateComplete state when either the guest flushes them or the threshold value is reached. The hypervisor changes the state of all “complete” buffers on the list to HvEventLogBufferStateReady and sends the list to the guest with a notification message. While the hypervisor may transition buffers on the completed list to the ready state at any time, the content of the buffer (aside from the state) will not change.
HvEventLogBufferStateReady – The buffer is complete (as with the HvEventLogBufferStateComplete state above) and has been sent to the guest with a notification message. Buffers in the “ready” state are under the exclusive control of the guest. Buffers leave the HvEventLogBufferStateReady state when the guest either unmaps them or releases them back to the hypervisor for reuse.

Note that the internal “standby” state is absent in the above description because it is not visible from the guest. While in the standby state, the buffer is not mapped, making the state information inaccessible. When in the “standby” state, the guest may either map the buffer (transitioning it to the HvEventLogBufferStateFree state) or delete the buffer.

Figure 2 depicts how buffers transition between states. Items labeled “guest” are guest actions (a hypercall), and those labeled “hypervisor” are actions taken by the hypervisor as event logging proceeds.

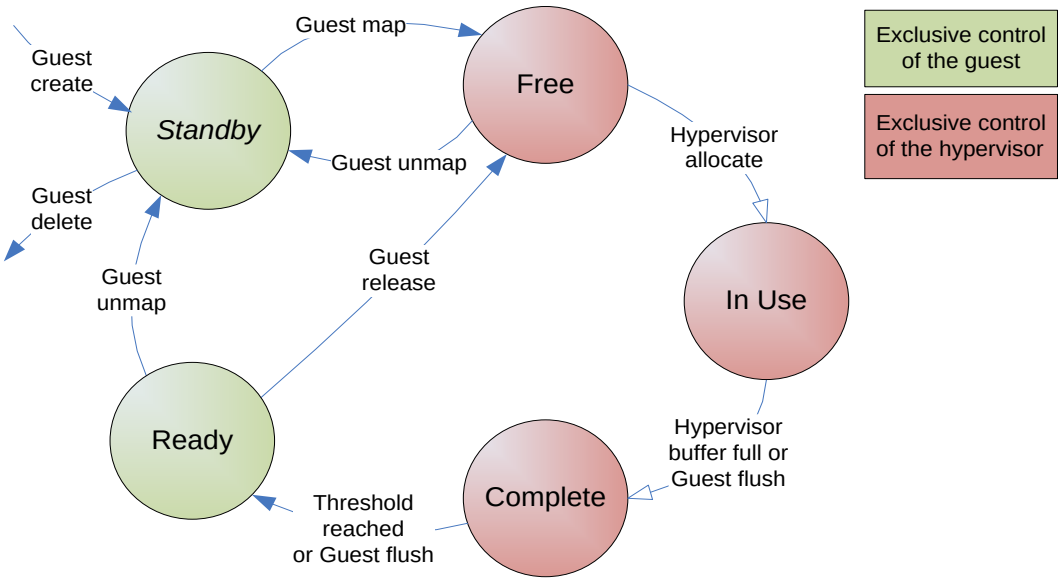


Figure 2 Event Log Buffer States

The following table summarizes the event logging operations that can be performed based upon the state of the buffer.

	Standby	Free	In Use	Complete	Ready
Initialize Buffer Group	N/A	N/A	N/A	N/A	N/A
Create Buffer	N/A	N/A	N/A	N/A	N/A
Map Buffer	Places the buffer into the free state	Error	Error	Error	Error

	Standby	Free	In Use	Complete	Ready
Request Flush	N/A	N/A	Places the buffer into the complete state, then …	Places buffers into the ready state and sends a message	N/A
Release Buffer	Error	Error	Error	Error	Places the buffer into the free state
Unmap Buffer	Error	Places the buffer into the standby state	Error	Error	Places the buffer into the standby state
Delete Buffer	Deallocates the buffer	Error	Error	Error	Error
Finalize Buffer Group	N/A	N/A	N/A	N/A	N/A

19.1.6 Accessing Event Log Buffers

Buffers are allocated using GPA pages deposited in the root partition's pool. When successfully created, a set of GPA pages are formed into a single, yet inaccessible buffer. The guest obtains the list of the buffer's GPA pages and enables read-only access to those pages by mapping the buffer. The buffer's pages remain read-only accessible until the buffer is successfully unmapped.

19.1.7 Preparing for Event Logging

The hypervisor supports a fixed set of event log types, each using a specific event log buffer class. When preparing to use an event logging type, the guest should consider the class, the expected rate of event generation and the NUMA topology when deriving the event log group's geometry and threshold.

The guest should set up the event logging environment for the type as follows:

- An adequate number of pages should be available in the partition's pool for both the logging infrastructure as well as the buffers themselves. If the local buffer class will be used, there should be adequate pages available for each NUMA proximity.
- The event log buffer group should be initialized. The number of buffers, the size of each buffer and the completed buffer threshold will be established. The number of buffers is a maximum value. There is no requirement that all buffers are created before enabling event logging for the type and buffers can be added to the group while it is active.
- An adequate supply of buffers should be created. If the local buffer class will be used, the buffers should be properly distributed across all NUMA proximities, based upon logical processor/proximity relationships.

The buffers should be mapped for read-only access. The hypervisor will not use a buffer until it has been mapped. In addition, the guest does not know where the buffer's pages are located until they have been mapped.

Once event logging has been set up, the guest may proceed to enable the logging of events.

19.1.8 Enabling and Disabling Event Logging

Associated with each event log type is a 64-bit *event source mask* that indicates which event sources are enabled. This gives the root partition fine-grained control over which events are of interest. The meaning of the bits within the mask is specific to the event log type.

Event logging is enabled by setting the relevant bits in the event source mask. This selects the set of sources whose events are to be recorded. In order to enable the event logging type, at least one source must be enabled; that is, the event mask must be non-zero.

Event logging is disabled by clearing all bits in the event source mask. This disables the recording of events from all sources for the event log type. The root partition may subsequently re-enable event logging or may finalize (terminate) event logging altogether.

19.1.9 Logging Events into Buffers

When an event log type is successfully enabled, logging from the specified event sources can begin. As logging proceeds, free buffers are selected on demand from the event log buffer group as the active buffer and placed in the "in use" state. For types that use the global buffer class, at most one buffer will be active at any time. For types that use the local buffer class, at most one buffer per logical processor will be "in use" at any time. Local buffers will be selected based upon its logical processor. When any "in use" buffer cannot hold another entry (or active buffers are flushed by the guest), it is placed in the "complete" state. When the number of buffers in this state reaches the threshold value defined at initialization time, the hypervisor places the buffers into a "ready" state and sends an "event log buffers ready" message to the guest. The message passes a list of buffers that are ready for the guest to examine. Note that the list may contain more buffers than the threshold value setting.

A buffer header will be present at the start of each buffer, possibly followed by a series of event log entries. These entries are added to the buffer in a packed, sequential manner. There may be unused space at the end of the buffer. The buffer remains in the control (read-only) of the guest until it is released back into the buffer group to be placed on the free list for reuse.

If the hypervisor is unable to obtain a free buffer when an event is ready to be recorded, the event will be lost. A mechanism to detect the loss of event messages is provided.

19.1.10 Event Log Buffers Ready Notification

During normal event log operation, buffers transition to the *HvEventLogBufferStateComplete* state and are placed at the end of a *pending buffer list,* awaiting guest notification. Buffers of the global class are placed onto a single, global list. Buffers of the local class are placed onto a per-logical processor list.

Buffers may be placed into the *HvEventLogBufferStateComplete* state for either of the following reasons:

- The buffer does not contain enough free space to record one more event, or
- The caller has requested that the active buffers for the type be flushed.

When either the count of buffers on a pending buffer list reaches the threshold value or buffers are explicitly flushed by the guest, the hypervisor transitions the buffers to the "ready state and

posts an “event log buffers ready” notification message to SINT0 of one of the root partition’s virtual processors. For event log types that use the local buffer class, the virtual processor chosen corresponds with the logical processor that produced the pending buffer list (root virtual processors have a hard affinity with logical processors, as described in section 22.4). Lists of pending global class buffers will be directed to any available virtual processor.

The message posted by the hypervisor includes both the event log type and the index of the first buffer in a *notification buffer list*. Buffers forwarded to the guest on the notification list are removed from the pending list and the count of pending buffers is reset to zero. Note that the number of actual buffers present on a notification list may exceed the threshold value. The hypervisor attempts to report as many ready buffers as possible with each message. Buffer lists are discussed in detail in section 19.1.11.

The guest is expected to respond to the message by reading the content of each of the buffers (perhaps writing the data to disk) and then releasing each of them back to the hypervisor so they will be returned to the free list and reused. Buffers on a notification list may be released to the hypervisor in any order. For a detailed description of the notification message format, see section 16.5.10.

19.1.11 Completed Buffer Lists

The ready buffers message generated by the hypervisor forwards a list of pending buffers to the guest. The message contains the event log buffer type and the index of the first buffer in the list. The *NextBufferIndex* field of the event log buffer header contains the index of the next buffer in the list. Buffers appear in the order in which they were appended to the list. The last buffer of the list contains the value HV_EVENTLOG_BUFFER_INDEX_NONE, indicating the end of the list.

19.1.12 Buffer Access Restrictions

Once a buffer is mapped by the guest, it remains read-only accessible to the guest until it is subsequently unmapped. During this period, the buffer may be in one of several states (as discussed in section 19.1.5). The guest will be able to access buffers that are in the free, in-use, complete and ready states (buffers in the standby state are not mapped and are therefore not accessible).

Buffers in the free state can be allocated by the hypervisor at any time. A free buffer has no meaningful content and therefore is of no interest to the guest. A guest may wish to deallocate a free buffer. Deallocation is a two-step process that involves unmapping, then deleting the buffer. Unmapping the buffer transitions it from the free to the standby state, where it can no longer be allocated for use by the hypervisor. Note that the hypervisor may allocate a free buffer between the time the guest examines the state and attempts to unmap it.

In-use buffers are allocated by the hypervisor and are being filled with individual event entries. Entries are appended to the buffer until it is full, at which time it transitions from the in-use to the complete state. In-use buffers can also be flushed by the guest, placing them into the complete state. While in the in-use state, the buffer’s header is subject to modification at any time. Modification however, will only cause new entries to be appended to the buffer and header updates are performed after the event data is copied. The guest may examine in-use buffers at any time but should be aware that more event data could be appended at any time.

Complete buffers are placed onto a list awaiting guest notification. Buffers in this state no longer accept new entries and are therefore ready for examination. Note that the hypervisor could transition these buffers to the ready state at any time. The NextBufferIndex field of the header is subject to change at any time and should not be examined. The buffer’s content however, will not change.

Ready buffers have either been reported to the guest or notification is in progress (the message may be pending delivery). These buffers are in a guest-controlled state and the hypervisor cannot

perform any operations them until they are released. All fields of the buffers in the ready state that have been delivered to the guest are stable and can no longer be changed by the hypervisor.

Since buffers in the in-use and complete state can transition state at any time, it is recommended that guests limit their access to event log buffers in the ready state that have been reported to them using the “complete buffers notification” message (see section 16.5.10). Such buffers remain in the ready state until the exclusive control of the guest until it takes action to cause a change in state.

19.1.13 Adding and Removing Buffers While Event Logging is Active

The guest may add new buffers to the event log buffer group at any time. New buffers enter the internal “standby” state until they are mapped, at which time they enter service as an available free buffer.

The guest may likewise remove buffers from the event log buffer group, but must do so in a special sequence. Removing a buffer begins by unmapping it. The unmap event log buffer hypercall returns a buffer to the internal “standby” state thereby taking the buffer out of service. The buffer may only be unmapped when it is either in the “free” or “ready” state (a flush operation can be used to bring “in use” buffers to the “completed” state and, subsequently, all “completed” buffers to the “ready” state). An unmapped buffer can be deleted at any time.

19.1.14 Concluding Event Logging

The conclusion procedure for event logging of a specific type is symmetrical with the preparation procedure. The guest should finalize the event logging environment for the type as follows:

Disable all event sources by calling HvSetEventLogGroupSources, specifying the event source mask as zero.

Flush all active buffers by calling HvFlushEventLogBuffer. If the guest wishes to retain the data, it should wait for the event log buffers ready messages to be delivered. Note that if the local buffer class is being used, a message may be generated for each logical processor. The guest should release the buffers back to the hypervisor to be placed onto the free list by calling HvReleaseEventLogBuffer.

The guest waits until all buffers are free before proceeding.

All mapped buffers are unmapped by calling HvUnmapEventLogBuffer. This revokes the read-only access to the buffer’s pages.

All buffers are deleted by calling HvDeleteEventLogBuffer. Deleting the buffers returns their pages to the partition’s pool from whence they came.

The event log buffer group is finalized by calling HvFinalizeEventLogGroup. Any pages allocated for the group’s infrastructure are returned to the partition’s pool.

Once event logging is concluded, excess free pages may be withdrawn from the partition’s pool.

19.2 Event Logging Data Types

19.2.1 Event Log Types

The hypervisor event logging interfaces are designed to be extensible and support new types in the future. The supported types are defined by the following enumeration.

```
typedef enum
{
      HvEventLogTypeGlobalSystemEvents    = 0x00000000
      HvEventLogTypeLocalDiagnostics      = 0x00000001,
} HV_EVENTLOG_TYPE;
```

Event Log Type	Usage	Scope
Diagnostics	Various internal events that can be used for debugging and performance analysis.	Local
System Events	Security auditing, history recording, hardware failures and reconfiguration.	Global

19.2.2 Event Enable Flags

Event log events can be selectively enabled or disabled by using a 64-bit mask. The meanings of the bits within this mask depend on the event log type.

```
// Diagnostic event log type event masks
#define HV_EVENTLOG_ENABLE_DIAG_ADMIN           0x0000000000000001UI64
#define HV_EVENTLOG_ENABLE_DIAG_PERF_GENERAL    0x0000000100000000UI64

// System event log type event masks
#define HV_EVENTLOG_ENABLE_SYSTEM_SUCCESS       0x0000000000000001UI64
#define HV_EVENTLOG_ENABLE_SYSTEM_FAILURE       0x0000000000000002UI64
```

19.2.3 Event Log Buffer State

Each buffer associated with an enabled event log type is in one of three states, defined by the following enumeration.

```
typedef enum
{
      HvEventLogBufferStateStandby  = 0,
      HvEventLogBufferStateFree     = 1,
      HvEventLogBufferStateInUse    = 2,
      HvEventLogBufferStateComplete = 3,
      HvEventLogBufferStateReady    = 4
} HV_EVENTLOG_BUFFER_STATE;
```

When a buffer is in the *HvEventLogBufferStateStandby* state, it is unmapped and not available for use by the hypervisor (note that when in this state the guest cannot access the buffer and therefore cannot see this value set in the state field of the buffer's header). The buffer enters the *HvEventLogBufferStateFree* state when it is mapped, making it eligible for use. The hypervisor may decide to place it into *HvEventLogBufferStateInUse* state at any time. When a buffer is in *HvEventLogBufferStateInUse* state, the hypervisor may add new event log events to it. When a buffer is placed into the *HvEventLogBufferStateComplete* state, it will no longer be modified by the hypervisor and it will be placed onto the completed buffer list. When the number of buffers on the list reaches the completed buffer threshold, the hypervisor will place all buffers on the list in the *HvEventLogBufferStateReady* state, and notify the guest that it may exclusively read the set

of buffers. The message will indicate the start of the buffer list (the buffer header contains the index of the next buffer in the list). The guest should extract any information that it needs from the buffers as quickly and efficiently as possible and release them back to the hypervisor to avoid any potential loss of event data. Releasing a buffer places it in the *HvEventLogBufferStateFree* state, making it immediately available for reuse. Alternatively, the guest may take a buffer out of service by unmapping it, placing it in the *HvEventLogBufferStateStandby* state.

19.2.4 Event Log Buffer Index

Event log buffers for each type are assigned an event log buffer index that is used to identify specific buffers within the type.

```
typedef UINT32 HV_EVENTLOG_BUFFER_INDEX, *PHV_EVENTLOG_BUFFER_INDEX;

#define HV_EVENTLOG_BUFFER_INDEX_NONE 0xffffffff
```

19.2.5 Event Log Buffer Header

Each buffer begins with the following header. Event log entry data begins immediately after the buffer header.

```
typedef struct
{
      UINT32      BufferSize;
      HV_EVENTLOG_BUFFER_INDEX      BufferIndex;
      UINT32      EventsLost;
      UINT32      ReferenceCounter;
      HV_NANO100_TIME   TimeStamp;
      UINT64      Reserved1;
      UINT64      Reserved2;
      HV_LOGICAL_PROCESSOR_INDEX    LogicalProcessor;
      HV_EVENTLOG_BUFFER_STATE      BufferState;
      UINT64      NextBufferOffset;
      HV_EVENTLOG_TYPE  Type;
      HV_EVENTLOG_BUFFER_INDEX      NextBufferIndex;
      UINT64      Reserved3;
      UINT64      Reserved4[2];
} HV_EVENTLOG_BUFFER_HEADER;
```

BufferSize is the size of the buffer in bytes.

BufferIndex is the hypervisor-assigned index of the buffer within this event log type. It was returned to the caller when the buffer was created.

LogicalProcessor is used only for local buffers, where it indicates which logical processor is responsible for the buffer. This field is undefined when using global buffers and when the buffer state is “free”.

EventsLost is the count of events that have not been recorded since the last event log buffer was placed into the complete state. It indicates that there were no free buffers available for the hypervisor to record these events and they have therefore been lost. When buffers become available, this counter is reset to zero. For event log types that use local buffers, this value is maintained on a logical processor basis.

TimeStamp records the time at which the buffer entered the “in use” state. This value represents the number of 100ns time units that have elapsed since the hypervisor was started. It can be used to determine the buffer fill order (that is, which buffers were “complete” first).

EventCount is an increasing counter that represents the sequential event number of the first event in the buffer. For event log types that use local buffers, this value is maintained on a logical processor basis. This value may also wrap on overflow.

BufferState indicates whether the buffer is currently free, in use, or complete.

NextBufferOffset indicates the byte offset into the buffer where the next entry will be written. This field is undefined if the buffer is free. When a buffer is marked “in use”, this value is initially set to the size of the buffer header to indicate that the next entry will be placed immediately after it. The hypervisor guarantees that entries written before this offset, if any, are complete. In other words, this value is updated only after an event has been recorded within the buffer. This allows a guest to safely read the partial content of a buffer before it enters the complete state.

Type indicates the Event Log Type that the buffer is related to. The combination of *Type* and *BufferIndex* can be used to identify a specific buffer.

NextBufferIndex contains the event log buffer index of the next buffer in the completion list, or the value HV_EVENTLOG_BUFFER_INDEX_NONE if this is the last one. This field is only valid for buffers in the HvEventLogBufferStateReady state.

19.2.6 Event Log Entry Header

One or more event log entries immediately follow the buffer header of each complete event log buffer. Each entry is composed of an event log entry header and, optionally, related event data. The total size of the header and any data must be aligned (by padding) to an 8-byte boundary.

```
typedef enum
{
HV_EVENTLOG_ENTRY_TIME_REFERENCE    = 0,
HV_EVENTLOG_ENTRY_TIME_TSC          = 1
} HV_EVENTLOG_ENTRY_TIME_BASIS;

typedef union
{
HV_NANO100_TIME               Reference;
UINT64                        Tsc;
} HV_EVENTLOG_ENTRY_TIME;

typedef struct
{
      UINT32                        Context;
      UINT16                        Size;
      UINT16                        Type;
      HV_EVENTLOG_ENTRY_TIME        TimeStamp;
} HV_EVENTLOG_ENTRY_HEADER;
```

Context is an implementation-specific field available to the event logging type.

Size is the size of the entry and is expressed in bytes. It includes the header and any padding bytes required to place the next entry on an 8-byte boundary.

Type is the entry type, specific to the event log type. The definitions of the entry types can be found in 32

TimeStamp records the time at which the entry was placed into the buffer. The source of the value was specified when the event log buffer group was created.

19.3 Event Logging Interfaces

19.3.1 HvInitializeEventLogBufferGroup

The HvInitializeEventLogBufferGroup hypercall defines the geometry (that is, the size and count) and establishes a message completion threshold for the event log buffer group associated with a specific type.

Wrapper Interface

```
HV_STATUS
HvInitializeEventLogBufferGroup(
      __in    HV_EVENTLOG_TYPE      Type,
      __in    UINT32                MaximumBufferCount,
      __in    UINT32                BufferPages,
      __in    UINT32                Threshold,
      __in    HV_EVENTLOG_ENTRY_TIME_BASIS      TimeBasis,
      __in    HV_NANO100_TIME             SystemTime
      );
```

Native Interface

HvInitializeEventLogBufferGroup		
	Call Code = 0x0060	
➡ Input Parameters		
0	Type (4 bytes)	MaximumBufferCount (4 bytes)
8	BufferPages (4 bytes)	Threshold (4 bytes)
16	TimeBasis (4 bytes)	SystemTime (8 bytes)
24	SystemTime	Padding (4 bytes)

Description

HvInitializeEventLogBufferGroup establishes the buffer size, maximum number of buffers, completion threshold and time basis for the event log buffer group. It allocates pages necessary to support the event logging infrastructure for the type. Pages are allocated from the guest's pool.

Input Parameters

Type specifies the type of the associated group to be initialized.

MaximumBufferCount is the maximum number of buffers that can exist for the associated event log buffer group at any time. Valid buffer index values are between zero and *MaximumBufferCount*-1.

BufferPages is the number of pages that will constitute each buffer for the associated buffer group.

Threshold is the minimum number of buffers that must be completed before the hypervisor will post a buffer completion message.

TimeBasis is the source of the event header timestamp. The value HV_EVENTLOG_ENTRY_TIME_REFERENCE indicates that the source is the hypervisor's reference time (time since the hypervisor was started). The value HV_EVENTLOG_ENTRY_TIME_TSC indicates that the source is the TSC of the logical processor posting the event to the buffer.

SystemTime is the current system time in 100 ns units.

Output Parameters

None.

Restrictions

The caller must be the root partition.

The event log type must not be previously initialized.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
	BufferPages specifies that buffers are composed of fewer than 1 or more than 512 pages.
	MaximumBufferCount specifies fewer than 1 or more than 512 buffers.
	The *Threshold* parameter specifies an inappropriate value.
	TimeBasis does not specify either HV_EVENTLOG_ENTRY_TIME_REFERENCE or HV_EVENTLOG_ENTRY_TIME_TSC.
HV_STATUS_OBJECT_IN_USE	The event log buffer group is already initialized.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the partition's memory pool is insufficient to perform the operation.

19.3.2 HvFinalizeEventLogBufferGroup

The HvFinalizeEventLogBufferGroup hypercall destroys the event log buffer group associated with the specified type.

Wrapper Interface

```
HV_STATUS
HvFinalizeEventLogBufferGroup(
      __in HV_EVENTLOG_TYPE   Type
      );
```

Native Interface

HvFinalizeEventLogBufferGroup [fast]		
	Call Code = 0x0061	
➡ Input Parameters		
0	Type (4 bytes)	Padding (4 bytes)

Description

The HvFinalizeEventLogBufferGroup hypercall is used to destroy the event log buffer group and return all allocated resources for the group's infrastructure to the partition's pool. The group should contain no buffers (all buffers should have been unmapped and deleted) when HvFinalizeEventLogBufferGroup is called or an error will be returned.

Input Parameters

Type specifies the event log type to be finalized.

Output Parameters

None.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
HV_STATUS_OPERATION_DENIED	There is at least one buffer remaining in the event log group associated with this type.
	The specified event log type has not been initialized.
	All sources for the specified event log type are not disabled.

19.3.3 HvCreateEventLogBuffer

The HvCreateEventLogBuffer hypercall allocates a single event log buffer for the specified event log buffer group.

Wrapper Interface

```
HV_STATUS
HvCreateEventLogBuffer(
      __in HV_EVENTLOG_TYPE   Type,
      __in HV_EVENTLOG_BUFFER_INDEX BufferIndex,
      __in HV_PROXIMITY_DOMAIN_INFO ProximityInfo
      );
```

Native Interface

HvCreateEventLogBuffer [fast]		
	Call Code = 0x0062	
➡ Input Parameters		
0	Type (4 bytes)	BufferIndex (4 bytes)
8	ProximityInfo (8 bytes)	

Description

The HvCreateEventLogBuffer hypercall is used to allocate an event log buffer for the specified event log buffer type. The size of the buffer must have previously been established by a call to HvInitializeEventLogBufferGroup. For types that use local buffers, the hypervisor selects pages for the buffer from the NUMA node described by *ProximityInfo*. The caller assigns the buffer a unique buffer index that may not already be in use and which must be between 0 and the maximum (less one) established at buffer group initialization time.

Input Parameters

Type specifies the event log type of the buffer group whose pages are to be mapped.

ProximityInfo specifies the NUMA proximity that the buffer's pages should be allocated from. This is mandatory when allocating local buffers and optional when the event log type uses global buffers. The Proximity Domain specified is described in section 7.2.1.

BufferIndex is the buffer index number to be assigned to the created buffer.

Output Parameters

None.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
	The specified proximity domain information is invalid.
	The specified buffer index is either already in use or is not within the limit as established at event log group initialization time.
HV_STATUS_OPERATION_DENIED	The event log buffer group associated with this type has not been initialized.
HV_STATUS_INSUFFICIENT_MEMORY	Insufficient memory exists for the call to succeed.

19.3.4 HvDeleteEventLogBuffer

The HvDeleteEventLogBuffer hypercall deallocates a single event log buffer for the specified event log buffer group.

Wrapper Interface

```
HV_STATUS
HvDeleteEventLogBuffer(
      __in HV_EVENTLOG_TYPE   Type,
      __in HV_EVENTLOG_BUFFER_INDEX BufferIndex
      );
```

Native Interface

HvDeleteEventLogBuffer [fast]		
	Call Code = 0x0063	
➡ Input Parameters		
0	Type (4 bytes)	BufferIndex (4 bytes)

Description

The HvDeleteEventLogBuffer hypercall is used to deallocate a specific event log buffer for the specified event log buffer type. The buffer must not be mapped. The pages that constitute the buffer are returned to the partition's pool from whence they came.

Input Parameters

Type specifies the event log type of the buffer group that this buffer is a member of.

BufferIndex is the buffer index number associated with the buffer to be deleted.

Output Parameters

None.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
	A buffer with the specified index does not exist.
HV_STATUS_OPERATION_DENIED	The event log buffer group has not been initialized.

Status code	Error condition
	The event log buffer associated with this index and type has not been unmapped.

19.3.5 HvMapEventLogBuffer

The HvMapEventLogBuffer hypercall returns the buffer's GPAs and read-only maps them into the guest's address space.

Wrapper Interface

```
HV_STATUS
HvMapEventLogBuffer(
      __in  HV_EVENTLOG_TYPE  Type,
      __in  HV_EVENTLOG_BUFFER_INDEX      BufferIndex,
      __out PHV_GPA_PAGE_NUMBER     GpaPages
      );
```

Native Interface

HvMapEventLogBuffer		
	Call Code = 0x0064	
➡ Input Parameters		
0	Type (4 bytes)	BufferIndex (4 bytes)
⬅ Output Parameters		
0	GpaPages (8 bytes) (Array of HV_GPA_PAGE_NUMBER values, the number determined by the *BufferPages* parameter of the HvIntializeEventLogBufferGroup hypercall)	

Description

The HvMapEventLogBuffer hypercall is used to read-only map the pages of the specified event log buffer into the guest's address space and to return the list of GPA pages that constitute the buffer. Successfully-mapped buffers are placed onto the free buffer list, making them available for the hypervisor to use for event logging.

Input Parameters

Type specifies the event log type of the buffer group whose pages are to be mapped.

BufferIndex is the index of the buffer within the buffer group associated with the specified event log type.

Output Parameters

GpaPages is an array of GPA page numbers that constitute the event log buffer. All buffers for a specific event log group are the same size. The size is established by the guest using the HvInitializeEventLogBufferGroup hypercall.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
	A buffer with the specified index does not exist.
HV_STATUS_OPERATION_DENIED	The event log buffer group has not been initialized.

Status code	Error condition
	The event log buffer associated with this index and type is already mapped.
HV_STATUS_INSUFFICIENT_MEMORY	Insufficient memory exists for the call to succeed.

19.3.6 HvUnmapEventLogBuffer

The HvUnmapEventLogBuffer hypercall unmaps an existing event log buffer.

Wrapper Interface

```
HV_STATUS
HvUnmapEventLogBuffer(
      __in    HV_EVENTLOG_TYPE      Type,
      __in    HV_EVENTLOG_BUFFER_INDEX    BufferIndex
      );
```

Native Interface

HvUnmapEventLogBuffer [fast]		
	Call Code = 0x0065	
➡ Input Parameters		
0	Type (4 bytes)	BufferIndex (4 bytes)

Description

HvUnmapEventLogBuffer revokes the read-only mappings of the pages that compose the specified buffer. Buffers can only be unmapped if they are in the free or ready state. The buffer is no longer available to be used by the hypervisor to log events.

Input Parameters

Type specifies the event log buffer type.

BufferIndex identifies the buffer to be unmapped.

Output Parameters

None.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
HV_STATUS_ACCESS_DENIED	The event log buffer is not in the free or ready state.

19.3.7 HvSetEventLogGroupSources

The HvSetEventLogGroupSources hypercall allows the caller to enable or disable specific event sources associated with the specific event log type.

Wrapper Interface

```
HV_STATUS
```

```
HvSetEventLogGroupSources(
      __in HV_EVENTLOG_TYPE   Type,
      __in UINT64 EnableFlags
      );
```

Native Interface

HvSetEventLogGroupSources [fast]		
	Call Code = 0x0066	
➡ Input Parameters		
0	Type (4 bytes)	Padding (4 bytes)
8	EnableFlags (8 bytes)	

Description

This hypercall allows the guest to enable and disable a set of up to 64 event sources. When *EnableFlags* is zero, event logging is disabled.

Input Parameters

Type specifies the type of the event log buffer.

EnableFlags is a 64-bit mask that specifies which event sources are enabled. The meaning of each bit in the mask depends on the event log type. In order for tracing to start, this mask must be non-zero; that is, at least one event source must be enabled.

Output Parameters

None.

Restrictions

The caller is not the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
HV_STATUS_OPERATION_DENIED	The event log buffer group is not initialized.

19.3.8 HvReleaseEventLogBuffer

The HvReleaseEventLogBuffer hypercall releases the completed buffer to the hypervisor as a free buffer.

Wrapper Interface

```
HV_STATUS
HvReleaseEventLogBuffer(
      __in HV_EVENTLOG_TYPE   Type,
      __in HV_EVENTLOG_BUFFER_INDEX BufferIndex
      );
```

Native Interface

HvReleaseEventLogBuffer [fast]		
	Call Code = 0x0067	
➡ Input Parameters		
0	Type (4 bytes)	BufferIndex (4 bytes)

Description

If the event log buffer specified is not in the “ready” state, an error will be returned. This call informs the hypervisor that the guest is finished processing the buffer. The hypervisor will place the buffer into a free state and make it available for reuse.

Input Parameters

Type specifies the type of the event log buffer.

BufferIndex specifies the index of the buffer within the event log type.

Output Parameters

None.

Restrictions

The caller must be the root partition.

The buffer specified by *BufferIndex* must be in the “complete” state.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
	The specified index for an event log buffer of this type is invalid.
HV_STATUS_OPERATION_DENIED	The event log buffer group has not been initialized.
	The specified event log buffer is not in the "ready" state.

19.3.9 HvFlushEventLogBuffer

The HvFlushEventLogBuffer hypercall forces an “in-use” event buffer to be marked as “complete” and an “event log buffers ready” notification to be queued.

Wrapper Interface

```
HV_STATUS
HvFlushEventLogBuffer (
      __in HV_EVENTLOG_TYPE   Type,
      __in HV_EVENTLOG_BUFFER_INDEX BufferIndex
      );
```

Native Interface

HvFlushEventLogBuffer [fast]		
	Call Code = 0x0068	
➡ Input Parameters		
0	Type (4 bytes)	BufferIndex (4 bytes)

Description

The HvFlushEventLogBuffer hypercall places an “in-use” event buffer into the “complete” state, as if the event buffer had filled up through event logging activity. It allows a caller to reclaim a buffer that is partially filled. The call also queues an “event log buffers ready” notification message to a VP within the root partition. This occurs even if the event buffer threshold has not yet been reached.

Input Parameters

Type specifies the type of the event log buffer.

BufferIndex specifies the index of the buffer within the event log type.

Output Parameters

None.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_INVALID_PARAMETER	The specified event log type is invalid.
	The specified index for an event log buffer of this type is invalid.
HV_STATUS_OPERATION_DENIED	The event log buffer group has not been initialized.
	The event log buffer is not in the "in use" state.

20 Guest Debugging Support

20.1 Overview

Many operating systems support inter-machine debugging that uses a physical connection to support communication between a debugger running on the *host* system and the component being debugged, running on the *target* system.. The most common approach is to use a NULL modem cable to connect a serial port on the host system to a serial port on the target system. Those ports are then used exclusively for debugging traffic. An alternative is to use an IEEE 1394 cable to connect the two systems. This chapter discusses both approaches, highlighting the differences where applicable.

When more than one guest is debugged simultaneously, each partition's traffic must be multiplexed across the common physical connection. The hypervisor provides a simple set of interfaces to exchange session-related data across the physical connection. The default port used for the connection is the serial COM port.

The implementation is almost entirely protocol independent. From the hypervisor's perspective, the debugging data exchanged by the two systems is simply a sequenced stream of opaque data. The exception is an additional multiplex message that translates into an NT debugger break-in message for the client. Non-NT debuggers do not need to handle this message for debugging to continue to work.

Note that because of the volatile nature of the connection, the hypervisor does not guarantee that any or all of the data will be delivered. The host and target systems should be aware of this and implement all appropriate synchronization and recovery mechanisms. Note that the hypervisor must have guest debugging support enabled. To detect the availability of the feature, use a hypervisor CPUID leaf. For details see section 3.3.

20.1.1 Debug Sessions

Each guest that uses the debugging facility manages a single *session* between itself and the external debugger. A partition must possess the *Debugging* privilege to use a debug session (see section 5.2.3). This privilege is granted at partition creation and checked when the partition is initialized to allow the system to reserve resources for the incoming and outgoing debugging data streams. Debugging resources are allocated from the partition's pool and provide buffers for incoming and outgoing session data. At a minimum, the outgoing buffer is large enough to accept the maximum amount of session data that can be posted by a single hypercall. The debugging privilege also grants access to the debugging hypercalls, HvPostDebugData, HvRetrieveDebugData and HvResetDebugSession.

The debug session is assigned a unique *debugging channel identifier* that identifies the session over the physical connection. The identifier is an assigned partition property value that is 16 bits wide and must either be zero or a non-zero value that is unique on the target system. A debugging channel identifier of zero indicates a channel ID is not assigned, and the debugging hypercalls return an error. For information regarding partition properties, see section 5.2.2.

When the partition is saved, any debugging-related data that has been posted or not yet received is not part of the partition's save context. When the partition is subsequently restored, the *Debugging* privilege must be granted at partition creation to ensure that the necessary session infrastructure is manifested by partition initialization. The session appears in a reset state, and the guest is expected to sense the loss of synchronization recover accordingly.

When the partition is finalized, the process implicitly resets the debugging session and releases any related resources that were allocated during partition initialization. Any debugging data in the incoming and outgoing buffers is discarded.

20.1.2 Transmitting Data on a Session

When a guest wishes to transmit data to an external debugger, it does so by *posting* the data to the session. The hypervisor sends the posted data to the host system asynchronously (for serial debugging) or synchronously (for 1394 debugging) to the host system. The hypervisor buffers the outgoing data using the buffer space that was allocated for the session when the partition was initialized. If sufficient free space is not available for all of the data, the hypervisor returns an error and does not post any data.

20.1.3 Retrieving Data from a Session

The hypervisor retrieves any outstanding received data for the session. The data is buffered using space that was allocated when the partition was initialized. If sufficient free space is not available when data arrives on the connection, the excess bytes are discarded.

The guest should request a buffer of the maximum size permitted by the hypercall. If the buffer has no data, the call returns an error. If the attempt to retrieve data is successful, more data might still be available. The caller should process the retrieved data and attempt to retrieve more. This process should be repeated until the hypercall returns an error indicating that all pending receive data has been retrieved.

20.1.4 Purging Data from a Session

If a guest detects a loss of synchronization with its remote counterpart, the guest can reset the session, selectively discarding all of the pending received or posted transmitted data for the session.

20.1.5 Sensing Activity on the Physical Connection

The hypervisor maintains a simple, global count of the number of bytes received over the debugging connection. The hypervisor increments this count each time a byte is received. The counter is permitted to wrap. A changing counter value is a good indication of activity, because it indicates that the session is active and the connection is intact. (Note that a changing counter value does not guarantee an active session, because the hypervisor could misconstrue noise as received data.)

The hypervisor also maintains a local, session-specific copy of the global counter that is updated from the global counter whenever the guest attempts to retrieve data from the session. If a guest wishes to test for incoming activity, it can do so by setting the appropriate options when it calls the post or retrieve hypercalls. If the options are set, the hypercall compares the local and global counters before doing anything else. If the two values match, then no receive activity has occurred on the physical connection since the local counter was last updated, and the hypercall returns an error. Guests can use this feature to detect inactivity and implement a time-out mechanism.

20.1.6 Hypercall Loops

During a debugging session, guests often enter tight loops as they attempt to post or retrieve session data until they are successful. The hypercalls used for this purpose allow the guest to provide a hint to the hypervisor that it intends to behave this way. The hypervisor is aware of the state of the debug session buffers and might elect to use the hint under either of the following circumstances:

- An attempt by the guest to retrieve data when the incoming data buffer is empty, or
- An attempt by the guest to post data when the outgoing data buffer has insufficient space.

When the hypervisor encounters either circumstance, it can use the hint and its knowledge of the connection and session state to do one of the following:

Return control to the guest, allowing it to make another attempt.

Suspend the virtual processor for the amount of time specified by the caller. The suspension could be terminated early if either of the following are true:

The guest is retrieving debugging data and has received new incoming data.

The guest is posting debugging data and adequate free outgoing buffer space has become available.

20.2 Debugging Data Types

The following data types support the debugging facility.

20.2.1 Debug Types

The debugging interfaces exchange simple blocks of data between a guest and an external port. The size of the blocks is not fixed, but cannot exceed a maximum size. The options that are accepted by the hypercalls and that guests use to implement debugging are defined as follows:

```
// Debug channel identifier
typedef UINT16 HV_DEBUG_CHANNEL_IDENTIFIER;

// Maximum size of the payload
#define HV_DEBUG_MAXIMUM_DATA_SIZE 4088

// Debug options for all calls
typedef UINT32 HV_DEBUG_OPTIONS;

// Options flags for HvPostDebugData
#define HV_DEBUG_POST_LOOP                          0x00000001

// Options flags for HvRetrieveDebugData
#define HV_DEBUG_RETRIEVE_LOOP                      0x00000001
#define HV_DEBUG_RETRIEVE_TEST_ACTIVITY             0x00000002

// Options flags for HvResetDebugSession
#define HV_DEBUG_PURGE_INCOMING_DATA                0x00000001
#define HV_DEBUG_PURGE_OUTGOING_DATA                0x00000002
```

20.3 Debugging Interfaces

The following sections describe the debug facility hypercall interfaces.

20.3.1 HvPostDebugData

The HvPostDebugData hypercall posts a block of data for transmission over the connection.

Wrapper Interface

```
HV_STATUS
HvPostDebugData(
      __in     UINT32                      Count,
      __in     HV_DEBUG_OPTIONS            Options,
      __in_bcount(HV_DEBUG_MAXIMUM_DATA_SIZE, Count)
               PUINT8                      Data,
      __out    PUINT32                     PendingCount
    );
```

Native Interface

HvPostDebugData		
	Call Code = 0x0069	
➡ Input Parameters		
0	Count (4 bytes)	Options (4 bytes)
8	Data (up to HV_DEBUG_MAXIMUM_DATA_SIZE bytes)	
⬅ Output Parameters		
0	PendingCount (4 bytes)	Padding (4 bytes)

Description

Users call the HvPostDebugData hypercall to post a block of data. The block can be as large as HV_DEBUG_MAXIMUM_DATA_SIZE bytes in size. If the hypervisor's outgoing data buffer has enough space for the entire block of data, it is posted for transmission. Otherwise, HvPostDebugData returns an error and no data is posted. Users can call HvPostDebugData with *Count* set to zero to determine whether any pending data is present in the outgoing session buffer.

Input Parameters

Count is the number of bytes to be posted. It can range from 0 to `HV_DEBUG_MAXIMUM_DATA_SIZE`.

Options specifies the following hypercall options:

HV_DEBUG_POST_LOOP indicates that the guest will continue to call HvPostDebugData until it completes successfully.

Data is an opaque stream of bytes for the hypervisor to post for transmission.

Output Parameters

PendingCount is the total number of bytes in the outgoing session buffer that are pending transmission, including any data posted by this call.

Restrictions

The caller's partition must possess the *Debugging* privilege.

Return Values

Status code	Error condition
HV_STATUS_FEATURE_UNAVAILABLE	Guest debugging support is not present or not enabled in the hypervisor.
HV_STATUS_ACCESS_DENIED	The caller's partition does not possess the *Debugging* privilege.
HV_STATUS_OPERATION_DENIED	The partition's *Debug Channel Identifier* property has not been assigned a unique, non-zero value.
HV_STATUS_INVALID_PARAMETER	The *Count* parameter specifies more than HV_DEBUG_MAXIMUM_DATA_SIZE bytes of data.
	The area described by *Data* and *Count* crosses a page boundary.
	Options specified a flag other than HV_DEBUG_POST_LOOP.
HV_STATUS_INSUFFICIENT_BUFFERS	The outgoing session buffer does not have sufficient space to accept all of *Data.*

20.3.2 HvRetrieveDebugData

The HvRetrieveDebugData hypercall retrieves received data.

Wrapper Interface

```
HV_STATUS
HvRetrieveDebugData(
      __in_out PUINT32        Count,
      __in     HV_DEBUG_OPTIONS     Options,
      __in     HV_NONA100_DURATION  Timeout,
      __out    PUINT32  RetrievedCount,
      __out    PUINT32  RemainingCount,
      __out_bcount(HV_DEBUG_MAXIMUM_DATA_SIZE, *Count)
               PUINT8         Data
      );
```

Native Interface

HvRetrieveDebugData		
	Call Code = 0x006A	
➡ Input Parameters		
0	Count (4 bytes)	Options (4 bytes)
8	Timeout (8 bytes)	
⬅ Output Parameters		
0	RetrievedCount (4 bytes)	RemainingCount (4 bytes)
8	Data (up to HV_DEBUG_MAXIMUM_DATA_SIZE bytes)	

Description

HvRetrieveDebugData retrieves up to HV_DEBUG_MAXIMUM_DATA_SIZE bytes from the session's received data stream and returns an error if no data is available. A caller can set *Count* to zero to simply determine whether any received data is present in the incoming session buffer. If data is present, HvRetrieveDebugData returns a success code but does not return any data.

Input Parameters

Count specifies the maximum number of bytes to be retrieved. This value can range from 0 to HV_DEBUG_MAXIMUM_DATA_SIZE.

Options specifies the hypercall options:

HV_DEBUG_RETRIEVE_LOOP indicates that if there is no immediately available data, the hypervisor should suspend the virtual processor for the interval specified by *Timeout*.

HV_DEBUG_RETRIEVE_TEST_ACTIVITY indicates that if no receive activity (for any session) has occurred on the physical connection since the last HvRetrieveDebugData hypercall, the hypercall should return an error. HvRetrieveDebugData performs this test before it does any other processing.

Timeout specifies the maximum time, in 100 nanosecond units, that the virtual processor will be suspended while waiting for data to arrive.

Output Parameters

RetrievedCount contains the number of bytes returned in *Data*.

RemainingCount contains the number of bytes in *Data* that are still pending, after *Count* bytes have been returned.

Data contains RetrievedCount bytes data from the beginning of the session's received data stream.

Restrictions

The caller's partition must possess the *Debugging* privilege.

Return Values

Status code	Error condition
HV_STATUS_FEATURE_UNAVAILABLE	Guest debugging support is not present or not enabled in the hypervisor.
HV_STATUS_ACCESS_DENIED	The caller's partition does not possess the *Debugging* privilege.
HV_STATUS_OPERATION_DENIED	The partition's *Debug Channel Identifier* property has not been assigned a unique, non-zero value.
HV_STATUS_INVALID_PARAMETER	The *Count* parameter specifies more than HV_DEBUG_MAXIMUM_DATA_SIZE bytes of data.
	The area described by *Data* and *Count* crosses a page boundary.
	Options specified a flag other than HV_DEBUG_RETRIEVE_LOOP or HV_DEBUG_RETRIEVE_TEST_ACTIVITY.
HV_STATUS_INACTIVE	*Options* specified HV_DEBUG_RETRIEVE_TEST_ACTIVITY and the hypervisor has not recorded any receive activity on the physical connection since the guest's last call to HvRetrieveDebugData.
HV_STATUS_NO_DATA	The incoming session buffer has no pending data.

20.3.3 HvResetDebugSession

The HvResetDebugSession hypercall resets the incoming or outgoing buffer context.

Wrapper Interface

```
HV_STATUS
HvResetDebugSession(
      HV_DEBUG_OPTIONS                    Options
      );
```

Native Interface

HvResetDebugSession [fast]		
	Call Code = 0x006B	
➡ Input Parameters		
0	Options (4 bytes)	Padding (4 bytes)

Description

The HvResetDebugSession hypercall directs the hypervisor to purge all internal receive or transmit buffers for the session. Any pending data is lost. Callers use this hypercall to reset the state of the connection.

Input Parameters

Options specifies which session buffers should be purged. Callers can set either or both of the following flags:

HV_DEBUG_PURGE_INCOMING_DATA purges the receive buffer.

HV_DEBUG_PURGE_OUTGOING_DATA purges the transmit buffer.

Output Parameters

None.

Restrictions

The caller's partition must possess the *Debugging* privilege.

Return Values

Status code	Error condition
HV_STATUS_FEATURE_UNAVAILABLE	Guest debugging support is not present or not enabled in the hypervisor.
HV_STATUS_ACCESS_DENIED	The caller's partition does not possess the *Debugging* privilege.
HV_STATUS_OPERATION_DENIED	The partition's *Debug Channel Identifier* property has not been assigned a unique, non-zero value.
HV_STATUS_INVALID_PARAMETER	The *Options* parameter includes an unsupported flag or includes neither of the supported flags.
HV_STATUS_NO_DATA	No data was purged from the buffers specified by the *Options* flags.

21 Statistics

21.1 Overview

The hypervisor maintains simple statistics counters for various events. The list of events is implementation-specific and likely to differ from one revision to the next. The following definitions help to set the context for further reading:

A *counter* monitors a specific item of the system.

An *object* is a collection of counters that are related to the same measurable entity.

A *counter instance* refers to a particular instantiation of an object that a counter can monitor.

21.1.1 Global Versus Local Statistics

Two classes of statistics counters are provided: global and local. Counters of the local class are local to a specific partition or objects associated with a partition. Counters of the global class are associated with the entire hypervisor or objects that are not associated with a specific partition.

Global statistics classes include:

- Hypervisor
- Logical processors

Local statistics classes include:

- Partitions
- Virtual processors

Additional objects for which counters of global and local classes are gathered may be added in future hypervisor implementations.

21.1.2 Statistics Page Mappings

Memory for statistics counter instances is allocated automatically by the hypervisor when certain objects are allocated. For example, when a virtual processor is allocated, space for that virtual processor's associated statistics is also allocated.

One page of memory is allocated to store the counter instances for each object and is called a *stats page*. This allows individual counter types to be mapped into a partition's address space for easy retrieval. This mapping is performed using overlay pages. For a discussion of overlay pages, see section 8.1.3.

Stats pages mapped into a partition's GPA space are write protected. Attempts to write to these pages will result in a #MC fault. The hypervisor writes to these pages by using writeback cacheability. For correct behavior, guests are recommended to map these pages with writeback caching as well.

The lifetime of a stats page is the same as the lifetime for its corresponding object. If that object is deleted and its stats page is still mapped within a partition's GPA space, that stats page mapping is removed. As with any overlay page, removal of a stats page mapping reveals whatever is "beneath" the overlay page.

21.1.3 Format of Statistics Pages

Statistics pages consist of a sequence of tightly-packed *statistics groups*, each prefixed by a header:

```
typedef UINT16 HV_STATISTICS_GROUP_TYPE;

typedef union
{
      UINT32      VersionMajorMinor;
      struct
      {
            UINT32    VersionMinor;
            UINT32    VersionMajor;
      } Version;
} HV_STATISTICS_GROUP_VERSION;

typedef UINT16 HV_STATISTICS_GROUP_LENGTH;

// Actual header format
typedef struct
{
      HV_STATISTICS_GROUP_TYPE      Type;
      HV_STATISTICS_GROUP_VERSION   Version;
      HV_STATISTICS_GROUP_LENGTH    Length;
} HV_STATISTICS_GROUP_HEADER;
```

Each statistics group has an associated identifier (TYPE) and version number present within its header. The group identifier is unique within the statistics object. The version number is composed of a major and minor number. The length field indicates how many bytes constitute the group. It does not include the size of the group header and is aligned to an 8-byte boundary. Adding the length to the base address of the group's data will lead to the next group header.

The last group of data within the page is followed by a header indicating the end of the list, with both the version number and length set to zero. Using this format allows the hypervisor to support a variety of both architectural and implementation-specific counters. See Appendix J: Statistics Counter Definitions for details of the various statistics group formats.

21.1.4 Statistics Group Compatibility

As new versions of the hypervisor are released and counters are added to the end of a group, the minor number will be incremented. This will allow existing code to be compatible with future hypervisors. A check to ensure that the group's major version number matches and that the minor version number is greater-than-or-equal to the compile-time value will suffice.

Should the format of the group ever change, the major number should be incremented. Note that groups containing architectural counters should not change major version numbers from release to release. A particular counter will always be present at its assigned location but may no longer be maintained. In such cases the counter will always read as zero.

```
// Standard header types that have zero version and length
#define HV_STATISTICS_TYPE_END_OF_LIST    0
#define HV_STATISTICS_TYPE_END_OF_PAGE    1

// Definitions for the hypervisor counters statistics page
#define HV_STATISTICS_TYPE_HCA_ID         2
#define HV_STATISTICS_TYPE_HCA_VERSION    ((1 << 16) | 0)   // V1.0

// Definitions for the logical processor counters statistics page
#define HV_STATISTICS_TYPE_LPA_ID         2
#define HV_STATISTICS_TYPE_LPA_VERSION    ((1 << 16) | 0)   // V1.0
#define HV_STATISTICS_TYPE_LPV_ID         3
#define HV_STATISTICS_TYPE_LPV_VERSION    ((1 << 16) | 0)   // V1.0
```

```
// Definitions for the partition counters statistics page
#define HV_STATISTICS_TYPE_PA_ID          2
#define HV_STATISTICS_TYPE_PA_VERSION     ((1 << 16) | 0)   // V1.0

// Definitions for the virtual processor statistics page
#define HV_STATISTICS_TYPE_VPA_ID         2
#define HV_STATISTICS_TYPE_VPA_VERSION    ((1 << 16) | 0)   // V1.0
#define HV_STATISTICS_TYPE_VPV_ID         3
#define HV_STATISTICS_TYPE_VPV_VERSION    ((1 << 16) | 0)   // V1.0
```

21.2 Statistics Data Types

21.2.1 Specifying Statistics Objects

When a caller wishes to access the statistics for an object, it must specify the object type and identify the object instance with the mapping request. The object type is specified using the following enumeration:

```
typedef enum
{
      // Global stats objects
      HvStatsObjectHypervisor = 0x00000001,
      HvStatsObjectLogicalProcessor = 0x00000002,

      // Local stats objects
      HvStatsObjectPartition  = 0x00010001,
      HvStatsObjectVp   = 0x00010002

} HV_STATS_OBJECT_TYPE;
```

The specific statistics object instance is identified using a 16-byte data structure, defined as follows:

```
typedef union
{
      // HvStatsObjectHypervisor
      struct
      {
      UINT64      ReservedZ1[2];
      } Hypervisor;

      // HvStatsObjectLogicalProcessor
      struct
      {
      HV_LOGICAL_PROCESSOR_INDEX    LogicalProcessorIndex;
            UINT32      ReservedZ2;
            UINT64      ReservedZ3;
      } LogicalProcessor;

      // HvStatsObjectPartition
      struct
      {
      HV_PARTITION_ID   PartitionId;
            UINT64      ReservedZ4;
      } Partition;

      // HvStatsObjectVp
      struct
      {
```

```
      HV_PARTITION_ID   PartitionId;
            HV_VP_INDEX VpIndex;
            UINT32      ReservedZ5;
      } Vp;

} HV_STATS_OBJECT_IDENTITY;
```

The flags associated with requests for certain statistics objects are defined as follows:

```
// Flags used for specifying the stats object when making
// mapping/unmapping stats page hypercalls

typedef     UINT16      HV_STATS_OBJECT_FLAG;

#define     HvStatsObjectSelfStats              0x0001
```

21.3 Statistics Interfaces

21.3.1 HvMapStatsPage

The HvMapStatsPage hypercall maps the specified stats page into the caller's GPA space.

Wrapper Interface

```
HV_STATUS
HvMapStatsPage(
      __in HV_STATS_OBJECT_TYPE     StatsType,
      __in HV_STATS_OBJECT_IDENTITY ObjectIdentity,
      __in HV_GPA_PAGE_NUMBER       TargetGpaPage
      );
```

Native Interface

HvMapStatsPage		
	Call Code = 0x006C	
➡ Input Parameters		
0	StatsType (4 bytes)	Padding (4 bytes)
8	ObjectIdentity (16 bytes)	
16		
24	TargetGpaPage (8 bytes)	

Description

Stats pages for global statistics can be mapped only into the GPA space of the root partition.

If the specified GPA page is beyond the bounds of the partition's valid GPA space, then the call may indicate success, but the stats page will not be accessible. In addition, the partition must have the *AccessStats* privilege.

Input Parameters

StatsType specifies the statistics object type.

ObjectIdentity specifies the statistics object's identity by using a union. The specific fields within this union differ depending on the statistics type, as specified in the first parameter. Reserved fields within this parameter must be set to zero.

When *StatsType* is HvStatsObjectHypervisor, *ObjectIdentity* is formatted as follows:

8	RsvdZ (16 bytes)
16	

When *StatsType* is HvStatsObjectLogicalProcessor, *ObjectIdentity* is formatted as follows:

8	LogicalProcessorIndex (4 bytes)	RsvdZ (4 bytes)
16	RsvdZ (8 bytes)	

When *StatsType* is HvStatsObjectPartition, *ObjectIdentity* is formatted as follows:

8	PartitionId (8 bytes)		
16	RsvdZ (8 bytes)	StatsObjectFlag (2 bytes)	RsvdZ (2 bytes)

When *StatsType* is HvStatsObjectVp, *ObjectIdentity* is formatted as follows:

8	PartitionId (8 bytes)		
16	VpIndex (4 bytes)	StatsObjectFlag (2 bytes)	RsvdZ (2 bytes)

TargetGpaPage specifies the location within the caller's GPA space where the stats page should appear.

Output Parameters

None.

Restrictions

The caller must possess the *AccessStats* privilege.

For global stats types, the caller must be the root partition.

For local stats types, the caller must be the parent of the partition specified by *PartitionId*.

The partition specified by *PartitionId* must be in the "active" state.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	For global stats types, the caller is not the root partition. For local stats types, the caller is not the parent of the specified partition.
	The caller's partition privilege flag AccessStats is cleared.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid (used only for partition and VP stats).
HV_STATUS_INVALID_VP_INDEX	The specified VP index does not reference a virtual processor within the specified partition (used only for VP stats).
HV_STATUS_OPERATION_DENIED	The stats page has already been mapped within the caller's GPA space. It must be unmapped before it is mapped again.
HV_STATUS_INVALID_PARAMETER	The specified logical processor index is invalid (used only for logical processor stats).
	Reserved fields within the *ObjectIdentity* parameter are not set to zero.
HV_STATUS_INSUFFICIENT_MEMORY	The number of pages in the memory pool of the caller is insufficient to perform the operation.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the "active" state.

21.3.2 HvUnmapStatsPage

The HvUnmapStatsPage hypercall unmaps the specified stats page from the caller's GPA space.

Wrapper Interface

```
HV_STATUS
HvUnmapStatsPage(
      __in HV_STATS_OBJECT_TYPE     StatsType,
      __in HV_STATS_OBJECT_IDENTITY ObjectIdentity
      );
```

Native Interface

HvUnmapStatsPage		
	Call Code = 0x006D	
➡ Input Parameters		
0	StatsType (4 bytes)	Padding (4 bytes)
8	ObjectIdentity (16 bytes)	
16		

Description

Only the root partition may unmap global stats pages. Local stats pages may only be unmapped by the partition or its parent. In addition, the partition must have the *AccessStats* privilege.

Input Parameters

StatsType specifies the statistics object type.

ObjectIdentity specifies the statistics object's identity by using a union. The specific fields within this union differ depending on the statistics type, as specified in the first parameter. Reserved fields within this parameter must be set to zero.

When *StatsType* is HvStatsObjectHypervisor, *ObjectIdentity* is formatted as follows:

8	RsvdZ (16 bytes)
16	

When *StatsType* is HvStatsObjectLogicalProcessor, *ObjectIdentity* is formatted as follows:

8	LogicalProcessorIndex (4 bytes)	RsvdZ (4 bytes)
16	RsvdZ (8 bytes)	

When *StatsType* is HvStatsObjectPartition, *ObjectIdentity* is formatted as follows:

8	PartitionId (8 bytes)		
16	RsvdZ (8 bytes)	StatsObjectFlag (2 bytes(	RsvdZ (2 bytes)

When *StatsType* is HvStatsObjectVp, *ObjectIdentity* is formatted as follows:

8	PartitionId (8 bytes)		
16	VpIndex (4 bytes)	StatsObjectFlag (2 bytes)	RsvdZ (2 bytes)

Output Parameters

None.

Restrictions

The caller must possess the *AccessStats* privilege.

For global stats types, the caller must be the root partition.

For local stats types, the caller must be the parent of the partition specified by *PartitionId.* The partition specified by *PartitionId* must be in the “active” state.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	For global stats types, the caller is not the root partition. For local stats types, the caller is not the parent of the specified partition.
	The caller’s partition privilege flag AccessStats is cleared.
HV_STATUS_INVALID_PARTITION_ID	The specified partition ID is invalid (used only for partition and VP stats).
HV_STATUS_INVALID_VP_INDEX	The specified VP index does not reference a virtual processor within the specified partition (used only for VP stats).
HV_STATUS_INVALID_PARAMETER	The specified logical processor index is invalid (used only for logical processor stats).
	Reserved fields within the *ObjectIdentity* parameter are not set to zero.
HV_STATUS_INVALID_PARTITION_STATE	The specified partition is not in the “active” state.
HV_STATUS_OPERATION_DENIED	The specified page is not initialized or mapped.

21.3.3 Statistics Page Mapping MSRs

A guest wishing to access its own local statistics pages must use the following model-specific registers (MSRs).

MSR address	Register name	Function
0x400000E0	HV_X64_MSR_STATS_PARTITION_RETAIL_PAGE	Map the guest’s local partition-related retail statistics page
0x400000E1	HV_X64_MSR_STATS_PARTITION_INTERNAL_PAGE	Map the guest’s local partition-related internal statistics page
0x400000E2	HV_X64_MSR_STATS_VP_RETAIL_PAGE	Map the guest’s local virtual processor-related retail statistics page
0x400000E3	HV_X64_MSR_STATS_VP_INTERNAL_PAGE	Map the guest’s local virtual processor-related internal statistics page

Each of the above MSRs has the following format:

63:12	11:1	0
Statistics Page Base Address	RsvdP	Enable

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

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

Restrictions

The caller must possess the *AccessStatsMsr* privilege.

22 Booting

22.1 Overview

The mechanism by which the hypervisor is loaded and launched is not architecturally defined. Typically, it will involve a *boot agent* running within ROM-based or flash-based firmware, an on-disk boot loader, or a device driver. This agent is responsible for loading the hypervisor code image, laying it out in memory, gathering various boot-time parameters, and invoking the hypervisor's entry point in the environment specified by the hypervisor image. Some boot agent implementations may leverage specialized security hardware to perform a *measured* or *verified launch*.

22.2 Pre-boot Requirements

The hypervisor or its boot agent must validate that the following requirements are true before or during a successful hypervisor launch:

- All physical processors support the minimum feature set required by the hypervisor.
- All physical processors have the same number of cores and hyperthreads per core.
- The physical processors support virtualization extensions (for example, Intel® VT or AMD–V™).
- The physical processors are not running in monitor mode; that is, the hypervisor is not attempting to be launched on top of an existing hypervisor or VMM.

For a detailed list of the hypervisor pre-boot requirements, see Appendix E: Boot-time CPUID Feature Requirements.

22.3 Post-boot Conditions

Immediately after a successful hypervisor launch, control is returned to the boot agent or another component of guest code designated by the boot agent. In any case, this code will be running in a newly-created partition referred to as the *root partition.* At this time, the following conditions are guaranteed to exist:

- The hypervisor knows about all present and potential physical nodes. Each of these nodes has been assigned a unique index.
- The hypervisor knows about the memory access ratios between physical nodes.
- An SPA range has been created for every populated RAM range in the machine.
- All RAM SPA pages are mapped into the root partition's GPA space with an identity map. All pages that are in use by the hypervisor are mapped with no access rights. All other pages are mapped as readable, writable, and executable.
- The hypervisor knows about all existing and potential logical processors in the system. Each has been given a unique logical processor index. (*Potential* here refers to logical processors that can be hot-plugged at some point in the future.)
- A single partition exists (the root partition). It is in the active state.
- The root partition's memory pool may or may not be populated with some memory. Code within the root partition can use the hypercall HvGetMemoryBalance to determine how many pages are in the pool.
- One virtual processor is created in the root partition for each existing and potential logical processor in the system.
- None of the root partition's virtual processors are suspended.
- All virtual processors corresponding to *potential* logical processors are assumed to be in the "offline" power state. All other virtual processors are assumed to be in the "C0" power state.

- After a successful hypervisor boot, the root partition's virtual processors are left in the same initialization state as their corresponding logical processors before the boot. For example, if a processor was already initialized and executing code, it will continue to execute code after hypervisor boot. If a processor was in the "wait for SIPI" state, it will effectively remain in this state after hypervisor boot.
- The initial register state of a root partition's virtual processor is not architecturally specified. Typically, it will be derived from the register state of the boot agent before the hypervisor was launched.
- The root partition runs with default scheduling policy parameters (no reserve, an infinite cap, and a weight of 100).
- The hypercall MSRs are visible to the root partition, but the hypercall interface has not yet established within the root partition.
- The SynIC is not yet enabled within the root partition. No synthetic timers are enabled.
- The local APIC is enabled and located in the root partition's GPA space according to the IA32_APIC_BASE MSR of the first logical processor (the boot processor).
- If available (see section 3.4), the APIC MSRs are visible within the root partition.

22.4 Root Partition

The root partition is special in a number of ways. It is "born" with certain properties, capabilities, and privileges that non-root partitions do not possess. These include the following:

- The root partition has no parent partition. Any operations that can be performed only by a parent partition on its children cannot be performed on the root. This includes the following operations:
- Changes to partition state (HvInitializePartition, HvFinalizePartition and HvDeletePartition)
- Modification of partition properties (HvGetPartitionProperty and HvSetPartitionProperty)
- Changes to GPA mappings (HvMapGpaPages may only change access rights to an existing mapping; HvUnmapGpaPages is not allowed)
- Access to VA or GPA space (HvTranslateVirtualAddress, HvReadGpa, or HvWriteGpa)
- Installation of intercepts (HvInstallIntercept)
- Virtual processor creation and deletion (HvCreateVp and HvDeleteVp)
- Access to virtual processor register state (HvGetVpRegisters and HvSetVpRegisters)
- Virtualization of legacy interrupts (HvAssertVirtualInterrupt and HvClearVirtualInterrupt)
- Saving and restoring partition state (HvSavePartitionState and HvRestorePartitionState)
- Management of logical processor states (HvGetLogicalProcessorRunTime) and HvCallParkLogicalProcessors.
- The root partition, by default, can access all physical I/O ports. Other partitions cannot. Note that some hypervisor implementations may restrict access to some I/O ports for security reasons.
- The root partition can access non-virtualized MSRs that directly control the behavior of the hardware. Note that some hypervisor implementations may restrict access to some MSRs for security reasons.
- The virtual processors within the root partition have hard affinities; that is, each virtual processor is strongly bound to a corresponding logical processor.
- By default, all mapped GPAs within the root partition's GPA space are mapped to the same addresses within the SPA space. This is not necessarily true for overlay pages.
- All hardware interrupts are directed at virtual processors within the root partition. (Future versions of the hypervisor will remove this assumption.)

23 System Properties

23.1 Overview

System properties provide a generic way for the root partition to control aspects of the entire hypervisor. After a system property is set, its value is constant unless and until it is again modified by the root partition. System properties are identified by a 32-bit code. System property values are each 128 bits in size.

23.2 System Property Data Types

```
typedef enum
{
    HvSystemProperyPerfCounterConfiguration,
    HvSystemPropertyMax
} HV_SYSTEM_PROPERTY_CODE;

typedef union
{
    UINT64 GenericProperty[2];
    HV_PERF_COUNTER_CONFIGURATION PerfCounter;
} HV_SYSTEM_PROPERTY;
```

23.3 Performance Counter Configuration

The performance counter configuration property specifies the performance counter and the sampling period to use during profiling. Setting this property does not start or stop the profiler – instead the profiler is started and stopped using event log hypercalls. See section 19. Only one event can be set for a given profile run. If the profiler is running when a call is made to set the property, it will fail with an HV_STATUS_OPERATION_DENIED status.

```
//
// Profile sources that can be configured.
//

typedef enum
{
    HvProfileInvalid,
    HvProfileCyclesNotHalted,
    HvProfileCacheMisses,
    HvProfileBranchMispredictions
} HV_PROFILE_SOURCE;

typedef struct
{
HV_PROFILE_SOURCE ProfileSource;
UINT32 Reserved;
UINT64 Period;
} HV_PERF_COUNTER_CONFIGURATION;
```

23.4 System Property Interfaces

23.4.1 HvSetSystemProperty

The HvSetSystemPropery hypercall configures settings that affect the entire system or hypervisor. Such settings are not specific to a particular partition.

Wrapper Interface

```
HV_STATUS
HvSetSystemProperty(
      __in HV_SYSTEM_PROPERTY_CODE  PropertyCode,
    __in HV_SYSTEM_PROPERTY   PropertyValue
);
```

Native Interface

HvSetSystemProperty		
	Call Code = 0x006F	
➡ Input Parameters		
0	PropertyCode (4 bytes)	Padding (4 bytes)
8	PropertyValue (16 bytes)	

Description

This call provides a generic mechanism to set system properties.

Input Parameters

PropertyCode specifies the property the caller is interested in modifying.

PropertyValue specifies the new property value.

Output Parameters

None.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_UNKNOWN_PROPERTY	The specified PropertyCode is not a recognized property.
HV_STATUS_INVALID_PARAMETER	An attempt was made to modify a read-only partition property or the specified value is invalid for the specified PropertyCode.
HV_STATUS_OPERATION_DENIED	The operation is not allowed in the current state.

23.4.2 HvGetSystemProperty

The HvGetSystemPropery hypercall returns settings that affect the entire system or hypervisor. Such settings are not specific to a particular partition.

Wrapper Interface

```
HV_STATUS
HvGetSystemProperty(
      __in HV_SYSTEM_PROPERTY_CODE  PropertyCode,
     __out PHV_SYSTEM_PROPERTY PropertyValue
);
```

Native Interface

HvGetSystemProperty		
	Call Code = 0x0071	
➡ Input Parameters		
0	PropertyCode (4 bytes)	Padding (4 bytes)
⬅ Output Parameters		
0	PropertyValue (16 bytes)	

Description

This call provides a generic mechanism to get system properties.

Input Parameters

PropertyCode specifies the property the caller is interested in querying.

Output Parameters

PropertyValue specifies the property's current value.

Restrictions

The caller must be the root partition.

Return Values

Status code	Error condition
HV_STATUS_ACCESS_DENIED	The caller is not the root partition.
HV_STATUS_UNKNOWN_PROPERTY	The specified PropertyCode is not a recognized property.

24 Appendix A: Interface Guidelines

In the interest of providing a consistent interface for the hypervisor, the following guidelines should be honored for all interfaces:

Names of calls, types, structures, and constants should be clearly marked as being associated with the hypervisor. In general, this is done by prepending “Hv” or “HV”.

Names of calls, types, structures, and constants should be clear and not overly generic (for example, “HvSetFlag” is probably too generic).

All hypervisor types (including enumerations) must have explicitly-defined sizes and values.

Names of types and constants that are platform-specific should include such an indication (for example, by prepending with “HV_X64”).

In general, abbreviations should be avoided. However, if abbreviations are used, they should be used consistently throughout the interface. The following is a list of approved abbreviations:

CPU (central processing unit)

ID (identifier)

INFO (information)

FP (floating point)

GPA (guest physical address)

HV (hypervisor)

MAX (maximum)

MIN (minimum)

MSR (model-specific register)

RAM (random access memory)

SINT (synthetic interrupt source)

SPA (system physical address)

STATS (statistics)

SYNIC (synthetic interrupt controller)

VP (virtual processor)

In most cases, partition IDs and VP indices should not be implicit; that is, the interface should allow the caller to specify these values rather than assuming the caller wants to operate on the current partition or VP. There are several specific cases where this guideline is ignored.

No hypercalls should include a GPA address input parameter that points to additional input or output parameters. In other words, all parameters should be passed by value, not by reference.

In most cases, page numbers are preferable to addresses. This is especially true when the interface in question operates on memory with page-level granularity.

The verb “Set” should be used only for calls that set a specific value. Generally, such calls should have an accompanying “Get” call.

The name of rep hypercalls should indicate that they operate on multiple parameters (for example, HvSetVpRegisters instead of HvSetVpRegister).

Rep hypercalls should be used only in cases where there is a clear performance need. In other cases, simple hypercalls are preferred.

25 Appendix B: Hypercall Code Reference

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 thru 0x0007			Reserved for future use	--	
0x0008		✔	HvNotifyLongSpinWait	Any	UseHypercallForLongSpinWait
0x0009			HvParkLogicalProcessors	Any	CpuManagement
0x000A through 0x003F			Reserved for future use	--	
0x0040			HvCreatePartition	Any	CreatePartitions
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 / Self	AccessMemoryPool
0x0049	✔		HvWithdrawMemory	Parent / Self	AccessMemoryPool
0x004A			HvGetMemoryBalance	Parent / Self	AccessMemoryPool
0x004B	✔		HvMapGpaPages	Parent / Root	
0x004C	✔		HvUnmapGpaPages	Parent	
0x004D			HvInstallIntercept	Parent	
0x004E			HvCreateVp	Parent	
0x004F		✔	HvDeleteVp	Parent	
0x0050	✔		HvGetVpRegisters	Parent / Self	
0x0051	✔		HvSetVpRegisters	Parent	
0x0052			HvTranslateVirtualAddress	Parent	
0x0053			HvReadGpa	Parent	
0x0054			HvWriteGpa	Parent	
0x0055			HvAssertVirtualInterrupt	Parent	
0x0056		✔	HvClearVirtualInterrupt	Parent	
0x0057			HvCreatePort	Parent / Self	CreatePort
0x0058		✔	HvDeletePort	Parent /	

Call Code	Rep Call	Fast Call	Hypercall	Caller	Partition Privilege Required (if any)
				Self	
0x0059			HvConnectPort	Parent / Self	ConnectPort
0x005A			HvGetPortProperty	Parent / Self	
0x005B		✔	HvDisconnectPort	Parent / Self	
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[1]	AccessStats
0x006D			HvUnmapStatsPage	Parent1	AccessStats
0x006E	✔		HvCallMapSparseGpaPages	Parent / Root	
0x006F			HvCallSetSystemProperty	Root	ConfigureProfiler
0x0070			HvCallSetPortProperty	Parent / Self	CreatePort

[1] Only the root partition may map global statistics pages.

26 Appendix C: Hypercall Status Code Reference

The following is a table of all hypercall return codes.

Status Code	Status Name	Meaning
0x0000	HV_STATUS_SUCCESS	The operation succeeded.
0x0001		Reserved.[1]
0x0002	HV_STATUS_INVALID_HYPERCALL_CODE	The hypercall code was not recognized.
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. 1
0x0006	HV_STATUS_ACCESS_DENIED	The caller did not possess sufficient access rights to perform the requested operation. 1
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.
0x0012	HV_STATUS_INVALID_CONNECTION_ID	The specified connection ID is not unique or does not exist.
0x0013	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		Reserved

Status Code	Status Name	Meaning
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_SSE3_NOT_SUPPORTED	The processor feature SSE3 is not supported.
0x0021	HV_STATUS_PROCESSOR_FEATURE_LAHFSAHF_NOT_SUPPORTED	The processor feature LAHFSAHF is not supported.
0x0022	HV_STATUS_PROCESSOR_FEATURE_SSSE3_NOT_SUPPORTED	The processor feature SSSE3 is not supported.
0x0023	HV_STATUS_PROCESSOR_FEATURE_SSE4_1_NOT_SUPPORTED	The processor feature SSE4.1 is not supported.
0x0024	HV_STATUS_PROCESSOR_FEATURE_SSE4_2_NOT_SUPPORTED	The processor feature SSE4.2 is not supported.
0x0025	HV_STATUS_PROCESSOR_FEATURE_SSE4A_NOT_SUPPORTED	The processor feature SSE4a is not supported.
0x0026	HV_STATUS_PROCESSOR_FEATURE_SSE5_NOT_SUPPORTED	The processor feature SSE5 is not supported.
0x0027	HV_STATUS_PROCESSOR_FEATURE_POPCNT_NOT_SUPPORTED	The processor feature POPCNT is not supported.
0x0028	HV_STATUS_PROCESSOR_FEATURE_CMPXCHG16B_NOT_SUPPORTED	The processor feature CMPXCHG16B is not supported.
0x0029	HV_STATUS_PROCESSOR_FEATURE_ALTMOVCR8_NOT_SUPPORTED	The processor feature ALTMOVCR8 is not supported.
0x002A	HV_STATUS_PROCESSOR_FEATURE_LZCNT_NOT_SUPPORTED	The processor feature LZCNT is not supported.
0x002B	HV_STATUS_PROCESSOR_FEATURE_MISALIGNED_SSE_NOT_SUPPORTED	The processor feature MISALIGNED SSE3 is not supported.
0x002C	HV_STATUS_PROCESSOR_FEATURE_MMX_EXT_NOT_SUPPORTED	The processor feature MMX EXT is not supported.

Status Code	Status Name	Meaning
0x002D	HV_STATUS_PROCESSOR_FEATURE_3DNOW_NOT_SUPPORTED	The processor feature 3DNow is not supported.
0x002E	HV_STATUS_PROCESSOR_FEATURE_EXTENDED_3DNOW_NOT_SUPPORTED	The processor feature Extended 3DNow is not supported.
0x002F	HV_STATUS_PROCESSOR_FEATURE_PAGE_1GB_NOT_SUPPORTED	The processor feature PAHGE 1GB is not supported.
0x0030	HV_STATUS_PROCESSOR_CACHE_LINE_FLUSH_SIZE_INCOMPATIBLE	The processor's cache line flush size is not supported.
0x0031	HV_STATUS_PROCESSOR_FEATURE_XSAVE_NOT_SUPPORTED	The processor feature XSAVE is not supported.
0x0032	HV_STATUS_PROCESSOR_FEATURE_XSAVEOPT_NOT_SUPPORTED	The processor feature XSAVEOPT is not supported.
0x0033	HV_STATUS_PROCESSOR_FEATURE_XSAVE_LEGACY_SSE_NOT_SUPPORTED	The processor feature XSAVE Legacy SSE is not supported.
0x0034	HV_STATUS_PROCESSOR_FEATURE_XSAVE_AVX_NOT_SUPPORTED	The processor feature XSAVE AVX is not supported.
0x0035	HV_STATUS_PROCESSOR_FEATURE_XSAVE_UNKNOWN_FEATURE_NOT_SUPPORTED	The processor unknown XSAVE feature is not supported.
0x0036	HV_STATUS_PROCESSOR_XSAVE_SAVE_AREA_INCOMPATIBLE	The processor's XSAVE area is not supported.
0x0037	HV_STATUS_INCOMPATIBLE_PROCESSOR	The processor architecture is not supported.

[1] Status code is currently referenced by a released version of Microsoft Windows ®

27 Appendix D: Architectural CPUID

The table below contains a list of architected CPUID leaves and how they are virtualized by the hypervisor. *Passthrough* means that the hardware value is used and passed through to the guest. That value will be the same on all logical processors in the system.

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
0x00000000					
EAX	Maximum valid standard CPUID index	0	31	Minimum of the hardware value on the current logical processor and 0x00000006	Minimum of the hardware value across all logical processors and 0x00000006
EBX	Processor vendor string	0	31	Passthrough	Passthrough
ECX	Processor vendor string	0	31	Passthrough	Passthrough
EDX	Processor vendor string	0	31	Passthrough	Passthrough
0x00000001					
EAX	Stepping	0	3	Passthrough	Passthrough minimum stepping across all logical processors
	Base Model	4	7	Passthrough	Passthrough
	Base Family	8	11	Passthrough	Passthrough
	Processor Type	12	13	Passthrough	Passthrough
	RsvdZ	14	15	Cleared	Cleared
	Extended Model	16	19	Passthrough	Passthrough
	Extended Family	20	27	Passthrough	Passthrough
	RsvdZ	28	31	Cleared	Cleared
EBX	Miscellaneous Information				
	Brand Identifier	0	7	Passthrough	Passthrough value received from processor 0
	CL Flush size	8	15	Passthrough	Passthrough
	Maximum LPs in a physical package	16	23	Passthrough	If HT is disabled, set to number of cores. If HT enabled, passthrough.
	Initial APIC ID	24	31	Passthrough	Return value of the HvX64RegisterInitialApicId
ECX	Feature Flags / Identifiers				
	SSE3	0	0	Passthrough	Set if set on all, otherwise 0
	RsvdZ	1	2	Cleared	Cleared
	MONITOR	3	3	Cleared	Cleared
	DS-CPL	4	4	Cleared	Cleared
	VMX	5	5	Cleared	Cleared
	RsvdZ	6	6	Cleared	Cleared

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	EST	7	7	Passthrough	Cleared
	TM2	8	8	Passthrough	Cleared
	SSSE3	9	9	Passthrough	Set if set on all, otherwise 0
	CNXTID	10	10	Passthrough	Cleared
	RsvdZ	11	12	Cleared	Cleared
	CMPXCHG16B	13	13	Passthrough	Set if set on all, otherwise 0
	xTPR	14	14	Passthrough	Cleared
	PDCM	15	15	Cleared	Cleared
	RsvdZ	16	17	Cleared	Cleared
	DCA	18	18	Cleared	Cleared
	SSE4.1	19	19	Passthrough	Set if set on all, otherwise 0
	SSE4.2	20	20	Passthrough	Set if set on all, otherwise 0
	RsvdZ	21	22	Cleared	Cleared
	Population Count	23	23	Passthrough	Set if set on all, otherwise 0
	RsvdZ	24	30	Cleared	Cleared
	Hypervisor Present	31	31	Set	Set
EDX	Feature Information				
	FPU	0	0	Set	Set
	VME	1	1	Set	Set
	DE	2	2	Set	Set
	PSE	3	3	Set	Set
	TSC	4	4	Set	Set
	MSR	5	5	Set	Set
	PAE	6	6	Set	Set
	MCE	7	7	Set	Set
	CMPXCHG8B	8	8	Set	Set
	APIC	9	9	Set	Set
	RsvdZ	10	10	Cleared	Cleared
	SEP	11	11	Set	Set
	MTRR	12	12	Set	Set
	PGE	13	13	Set	Set
	MCA	14	14	Set	Set
	CMOV	15	15	Set	Set
	PAT	16	16	Set	Set
	PSE-36	17	17	Set	Set
	PSN	18	18	Cleared	Cleared
	CLFSH	19	19	Set	Set
	RsvdZ	20	20	Cleared	Cleared
	DS	21	21	Cleared	Cleared
	ACPI	22	22	Passthrough	Cleared
	MMX	23	23	Set	Set
	FXSR	24	24	Set	Set

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	SSE	25	25	Set	Set
	SSE2	26	26	Set	Set
	SS	27	27	Passthrough	Set if set on all, otherwise 0
	HTT	28	28	Passthrough	Cleared
	TM	29	29	Passthrough	Cleared
	RsvdZ	30	30	Cleared	Cleared
	PBE	31	31	Passthrough	Cleared
0x00000002					
EAX	Cache and TLB descriptors	0	31	Passthrough on Intel Cleared on AMD	Passthrough value from processor 0 on Intel Cleared on AMD
EBX	Cache and TLB descriptors	0	31	Passthrough on Intel Cleared on AMD	Passthrough value from processor 0 on Intel Cleared on AMD
ECX	Cache and TLB descriptors	0	31	Passthrough on Intel Cleared on AMD	Passthrough value from processor 0 on Intel Cleared on AMD
EDX	Cache and TLB descriptors	0	31	Passthrough on Intel Cleared on AMD	Passthrough value from processor 0 on Intel Cleared on AMD
0x00000003					
EAX	Processor serial number	0	31	Cleared	Cleared
EBX	Processor serial number	0	31	Cleared	Cleared
ECX	Processor serial number	0	31	Cleared	Cleared
EDX	Processor serial number	0	31	Cleared	Cleared
0x00000004					
EAX	Cache type	0	4	Passthrough	Passthrough value from processor 0
	Cache level	5	7	Passthrough	Passthrough value from processor 0
	Self-initializing cache level	8	8	Passthrough	Passthrough value from processor 0
	Fully Associative	9	9	Passthrough	Passthrough value from processor 0
	Write back invalidate / invalidate	10	10	Passthrough	Passthrough value from processor 0
	Cache inclusiveness	11	11	Passthrough	Passthrough value from processor 0
	RsvdZ	12	13	Cleared	Cleared

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	Maximum threads in cache	14	25	Passthrough	If HT disabled, scale the value down by the number of threads / core. If HT is enabled, set to the number of logical processors per package.
	Max cores per package	26	31	Passthrough	If HT enabled, scale up this value by number of threads per core. If HT is disabled, passthrough the value from processor 0.
EBX	Coherency line size	0	11	Passthrough	Passthrough value from processor 0
	Physical line partitions	12	21	Passthrough	Passthrough value from processor 0
	Ways of associativity	22	31	Passthrough	Passthrough value from processor 0
ECX	Number of sets	0	31	Passthrough	Passthrough value from processor 0
EDX	RsvdZ	0	31	Cleared	Cleared
0x00000005	MONITOR / MWAIT				
EAX	MonLineSizeMin	0	15	Cleared	Cleared
	RsvdZ	16	31	Cleared	Cleared
EBX	MonLineSizeMax	0	15	Cleared	Cleared
	RsvdZ	16	31	Cleared	Cleared
ECX	EMX	0	0	Cleared	Cleared
	IBE	1	1	Cleared	Cleared
	RsvdZ	2	31	Cleared	Cleared
EDX	C0SubstatesUsingMwait	0	3	Cleared	Cleared
	C1SubstatesUsingMwait	4	7	Cleared	Cleared
	C2SubstatesUsingMwait	8	11	Cleared	Cleared
	C3SubstatesUsingMwait	12	15	Cleared	Cleared
	C4SubstatesUsingMwait	16	19	Cleared	Cleared
	RsvdZ	20	31	Cleared	Cleared
0x00000006	Power management feature enumeration function				
EAX	Digital Temperature Sensor Supported.	0	0	Passthrough for the partition possessing the CpuManagement privilege, otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege, otherwise Cleared

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	Dynamic Acceleration Enabled	1	1	Passthrough for the partition possessing the CpuManagement privilege, otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
	Constant Rate Timer	2	2	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
	RsvdZ	3	31	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
EBX	Thermal Threshold Count	0	3	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
	RsvdZ	4	31	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
ECX	Hardware Coordination Feeback	0	0	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
	RsvdZ	1	31	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
EDX	RsvdZ	0	31	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared	Passthrough for the partition possessing the CpuManagement privilege , otherwise Cleared
0x00000007					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x00000008					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x00000009					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x0000000A					
EAX	Architectural Perfmon	0	31	Cleared	Cleared
EBX	Architectural Perfmon	0	31	Cleared	Cleared
ECX	Architectural Perfmon	0	31	Cleared	Cleared
EDX	Architectural Perfmon	0	31	Cleared	Cleared
0x40000000	Hypervisor CPUID leaf range and vendor ID signature				
EAX	The maximum input value for hypervisor CPUID information	0	31	At least 0x40000005	At least 0x40000005
EBX	Hypervisor Vendor ID Signature	0	31	0x7263694D—“Micr”	0x7263694D—“Micr”
ECX	Hypervisor Vendor ID Signature	0	31	0x666F736F—“osof”	0x666F736F—“osof”
EDX	Hypervisor Vendor ID Signature	0	31	0x76482074—“t Hv”	0x76482074—“t Hv”
0x40000001	Hypervisor vendor-neutral interface identification				
EAX	Hypervisor Interface Signature	0	31	0x31237648—“Hv#1”	0x31237648—“Hv#1”
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x40000002	Operating system identity				
EAX	Build Number	0	31		
EBX	Minor Version	0	15		
	Major Version	16	31		
ECX	Service Pack	0	31		

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
EDX	Service Number	0	23		
	Service Branch	24	31		
0x40000003	Feature Identification				
EAX	Feature identification - partition privileges				
	VP Runtime available	0	0		
	Partition Reference Counter available	1	1		
	Basic SynIC MSRs available	2	2		
	Synthetic Timer MSRs available	3	3		
	APIC access MSRs available	4	4		
	Hypercall MSRs available	5	5		
	Access virtual processor index MSR available	6	6		
	Virtual System Reset MSR available	7	7		
	Access statistics pages MSRs available	8	8		
	RsvdZ	9	31	Cleared	Cleared
EBX	Feature identification - partition creation flags				
	CreatePartitions	0	0		
	AccessPartitionId	1	1		
	AccessMemoryPool	2	2		
	AdjustMessageBuffers	3	3		
	PostMessages	4	4		
	SignalEvents	5	5		
	CreatePort	6	6		
	ConnectPort	7	7		
	AccessStats	8	8		
	RsvdZ	9	10		
	Debugging	11	11		
	CpuManagement	12	12		
	RsvdZ	13	31	Cleared	Cleared
ECX	Feature identification - power management				
	Maximum processor power state. 0 is C0, 1 is C1, 2 is C2, 3 is C3.	0	3		
	RsvdZ	4	31	Cleared	Cleared
EDX	Feature identification - miscellaneous				
	The MWAIT instruction is available	0	0		
	Guest debugging support is available	1	1		

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	Performance monitor support is available	2	2		
	RsvdZ	3	31	Cleared	Cleared
0x40000004	Implementation Recommendations				
EAX	Recommend using hypercall for address space switches rather than MOV to CR3 instruction	0	0		
	Recommend using hypercall for local TLB flushes rather than INVLPG or MOV to CR3 instructions	1	1		
	Recommend using hypercall for remote TLB flushes rather than inter-processor interrupts	2	2		
	Recommend using MSRs for accessing APIC registers EOI, ICR and TPR rather than their memory-mapped counterparts.	3	3		
	Recommend using the hypervisor-provided MSR to initiate a system RESET.	4	4		
	Recommend using relaxed timing for this partition.	5	5		
	RsvdZ	6	31	Cleared	Cleared
EBX	Recommended number of attempts to retry a spinlock failure before notifying the hypervisor about the failures. 0xFFFFFFFF indicates never to retry.	0	31		
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x40000005	Implementation limits				
EAX	The maximum number of virtual processors supported.	0	31		
EBX	The maximum number of logical processors supported.	0	31		
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x40000006	Implementation hardware features.				

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
EAX	Support for APIC overlay assist is detected and in use.	0	0		
	Support for MSR bitmaps is detected and in use.	1	1		
	Support for architectural performance counters is detected and in use.	2	2		
	Support for second level address translation is detected and in use.	3	3		
	RsvdZ for future Intel-specific features.	4	31		
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ for future AMD-specific features	0	31	Cleared	Cleared
0x40000007 - 0x4000007F					
EAX - EDX	Reserved for future hypervisor use				
0x40000080 - 0x400000FF					
EAX - EDX	Reserved for use of intercept handlers in the parent partition				
0x80000000					
EAX	Highest Extended CPUID Leaf	0	31	Minimum of the hardware value on current logical processor and 0x8000001A	Minimum of the hardware value across all logical processors in the system and 0x8000001A
EBX	Processor vendor string	0	31	Passthrough	Passthrough value from processor 0
ECX	Processor vendor string	0	31	Passthrough	Passthrough value from processor 0
EDX	Processor vendor string	0	31	Passthrough	Passthrough value from processor 0
0x80000001					
EAX	Stepping	0	3	Passthrough	Minimum across all logical processors
	Base Model	4	7	Passthrough	Passthrough value from processor 0
	Base Family	8	11	Passthrough	Passthrough value from processor 0
	Processor Type	12	13	Passthrough	Passthrough value from processor 0
	RsvdZ	14	15	Cleared	Cleared
	Extended Model	16	19	Passthrough	Passthrough

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	Extended Family	20	27	Passthrough	Passthrough
	RsvdZ	28	31	Cleared	Cleared
EBX	Brand ID	0	15	Passthrough	Passthrough
	RsvdZ	16	27	Cleared	Cleared
	Package Type	28	31	Passthrough	Passthrough
ECX	Extended feature flag / feature identifiers				
	LahfSahf	0	0	Passthrough	Set if set on all, otherwise 0
	CmpLegacy	1	1	Passthrough	Set if set on all, otherwise 0
	SVM	2	2	Cleared	Cleared
	ExtApicSpace	3	3	Cleared	Cleared
	AltMovCr8	4	4	Passthrough	Set if set on all, otherwise 0
	ABM	5	5	Passthrough	Set if set on all, otherwise 0
	SSE4A	6	6	Passthrough	Set if set on all, otherwise 0
	MisAlignSse	7	7	Passthrough	Set if set on all, otherwise 0
	3DNowPrefetch	8	8	Passthrough	Set if set on all, otherwise 0
	OSVW	9	9	Passthrough	Set if set on all, otherwise 0
	RsvdZ	10	10	Cleared	Cleared
	SSE5	11	11	Passthrough	Set if set on all, otherwise 0
	SKINIT	12	12	Cleared	Cleared
	WDT	13	13	Passthrough	Cleared
	RsvdZ	14	31	Cleared	Cleared
EDX	Extended feature flags				
	FPU	0	0	Set	Set
	VME	1	1	Passthrough	Set if set on all, otherwise 0
	DE	2	2	Set	Set
	PSE	3	3	Set	Set
	TSC	4	4	Set	Set
	MSR	5	5	Set	Set
	PAE	6	6	Set	Set
	MCE	7	7	Set	Set
	CMPXCHG8B	8	8	Set	Set
	APIC	9	9	Set	Set
	RsvdZ	10	10	Cleared	Cleared
	SysCallSysRet	11	11	Set	Set
	MTRR	12	12	Set	Set
	PGE	13	13	Set	Set
	MCA	14	14	Set	Set
	CMOV	15	15	Set	Set

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	PAT	16	16	Set	Set
	PSE36	17	17	Passthrough	Set if set on all, otherwise 0
	RsvdZ	18	18	Cleared	Cleared
	RsvdZ	19	19	Cleared	Cleared
	Execute disabled / No execute	20	20	Set	Set
	RsvdZ	21	21	Cleared	Cleared
	MmxExt	22	22	Passthrough	Set if set on all, otherwise 0
	MMX	23	23	Set	Set
	FXSR	24	24	Set	Set
	FFXSR	25	25	Passthrough	Set if set on all, otherwise 0
	Page1GB	26	26	Cleared	Cleared
	RDTSCP	27	27	Passthrough	Cleared
	RsvdZ	28	28	Cleared	Cleared
	LM	29	29	Set	Set
	3DNowExt	30	30	Passthrough	Set if set on all, otherwise 0
	3DNow	31	31	Passthrough	Set if set on all, otherwise 0
0x80000002	Processor name string identifier				
EAX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
EBX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
ECX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
EDX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
0x80000003	Processor name string identifier				
EAX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
EBX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
ECX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
EDX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
0x80000004	Processor name string identifier				
EAX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
EBX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
ECX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0
EDX	Processor Brand String	0	31	Passthrough	Passthrough value from processor 0

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
0x80000005	L1 Cache and TLB identifiers (all registers)				
EAX	L1ITlb2and4MSize	0	7	Passthrough	Passthrough
	L1ITlb2and4MAssoc	8	15	Passthrough	Passthrough
	L1DTlb2and4MSize	16	23	Passthrough	Passthrough
	L1DTlb2and4MAssoc	24	31	Passthrough	Passthrough
EBX	L1ITlb4KSize	0	7	Passthrough	Passthrough
	L1ITlb4KAssoc	8	15	Passthrough	Passthrough
	L1DTlb4KSize	16	23	Passthrough	Passthrough
	L1DTlb4KAssoc	24	31	Passthrough	Passthrough
ECX	L1DcLineSize	0	7	Passthrough	Passthrough
	L1DcLinesPerTag	8	15	Passthrough	Passthrough
	L1DcAssoc	16	23	Passthrough	Passthrough
	L1DcSize	24	31	Passthrough	Passthrough
EDX	L1IcLineSize	0	7	Passthrough	Passthrough
	L1IcLinesPerTag	8	15	Passthrough	Passthrough
	L1IcAssoc	16	23	Passthrough	Passthrough
	L1IcSize	24	31	Passthrough	Passthrough
0x80000006	L2 Cache and L2 TLB identifiers				
EAX	L2ITlb2and4MSize	0	11	Passthrough	Passthrough
	L2ITlb2and4MAssoc	12	15	Passthrough	Passthrough
	L2DTlb2and4MSize	16	27	Passthrough	Passthrough
	L2DTlb2and4MAssoc	28	31	Passthrough	Passthrough
EBX	L2ITlb4KSize	0	11	Passthrough	Passthrough
	L2ITlb4KAssoc	12	15	Passthrough	Passthrough
	L2DTlb4KSize	16	27	Passthrough	Passthrough
	L2DTlb4KAssoc	28	31	Passthrough	Passthrough
ECX	L2 Line Size	0	7	Passthrough	Passthrough
	L2 Lines per tag	8	11	Passthrough	Passthrough
	L2 Associativity	12	15	Passthrough	Passthrough
	L2 cache size in kilobytes	16	31	Passthrough	Passthrough
EDX	L3LineSize	0	7	Passthrough	Passthrough
	L3LinesPerTag	8	11	Passthrough	Passthrough
	L3Assoc	12	15	Passthrough	Passthrough
	RsvdZ	16	17	Passthrough	Passthrough
	L3Size	18	31	Passthrough	Passthrough
0x80000007	Advanced Power Management Information				
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	TS	0	0	Passthrough	Cleared
	FID	1	1	Passthrough	Cleared
	VID	2	2	Passthrough	Cleared
	TTP	3	3	Passthrough	Cleared
	TM	4	4	Passthrough	Cleared
	STC	5	5	Passthrough	Cleared
	100MhzSteps	6	6	Passthrough	Cleared
	HwPState	7	7	Passthrough	Cleared

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
	TscInvariant	8	8	Passthrough	Cleared
	RsvdZ	9	31	Cleared	Cleared
0x80000008					
EAX	PhysAddrSize	0	7	Passthrough	The size of the physical GPA space that is supported
	LinAddrSize	8	15	Passthrough	The size of the virtual GPA space that is supported
	RsvdZ	16	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	NC	0	7	Passthrough	Passthrough
	RsvdZ	8	11	Cleared	Cleared
	ApicIdCoreIdSize	12	15	Passthrough	Passthrough
	RsvdZ	16	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000009					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x8000000A	SVM revision and feature identification				
EAX	SvmRev	0	7	Cleared	Cleared
	RsvdZ	8	31	Cleared	Cleared
EBX	NASID	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	NP	0	0	Cleared	Cleared
	LBRVirt	1	1	Cleared	Cleared
	SVML	2	2	Cleared	Cleared
	NRIPS	3	3	Cleared	Cleared
	RsvdZ	4	31	Cleared	Cleared
0x8000000B					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x8000000C					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x8000000D					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x8000000E					

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x8000000F					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000010					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000011					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000012					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000013					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000014					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000015					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000016					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000017					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared

Index and Register	Name	Start bit	End bit	Virtualized Value Root	Virtualized Value Non-Root
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000018					
EAX	RsvdZ	0	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x80000019					
EAX	L1ITlb1GSize	0	11	Cleared	Cleared
	L1ITlb1GAssoc	12	15	Cleared	Cleared
	L1DTlb1GSize	16	27	Cleared	Cleared
	L1DTlb1GAssoc	28	31	Cleared	Cleared
EBX	L2ITlb1GSize	0	11	Cleared	Cleared
	L2ITlb1GAssoc	12	15	Cleared	Cleared
	L2DTlb1GSize	16	27	Cleared	Cleared
	L2DTlb1GAssoc	28	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared
0x8000001A					
EAX	FP128	0	0	Passthrough	Set if set on all, otherwise 0
	MOVU	1	1	Passthrough	Set if set on all, otherwise 0
	RsvdZ	2	31	Cleared	Cleared
EBX	RsvdZ	0	31	Cleared	Cleared
ECX	RsvdZ	0	31	Cleared	Cleared
EDX	RsvdZ	0	31	Cleared	Cleared

28 Appendix E: Boot-time CPUID Feature Requirements

The table below contains a list of CPUID values required for the hypervisor to boot.

Index and Register	Name	Start bit	End bit	Virtualized feature set definition between processors in the same system
0x00000000				
EAX	Maximum valid standard CPUID index	0	31	Hardware-specific If the underlying hardware is AMD-based, the value must be at least 0x00000001. If the underlying hardware is Intel-based, the value must be at least 0x00000004.
EBX	Processor vendor string	0	31	Must match
ECX	Processor vendor string	0	31	Must match
EDX	Processor vendor string	0	31	Must match
0x00000001				
EAX	Stepping	0	3	--
	Base Model	4	7	Must match
	Base Family	8	11	Must match
	Processor Type	12	13	Must match
	RsvdZ	14	15	--
	Extended Model	16	19	Must match
	Extended Family	20	27	Must match
	RsvdZ	28	31	--
EBX	Miscellaneous Information			
	Brand Identifier	0	7	--
	CL Flush size	8	15	Must match
	Maximum LPs in a physical package	16	23	Must match
	Initial APIC ID	24	31	--
ECX	Feature Flags / Identifiers			
	SSE3	0	0	--
	RsvdZ	1	2	--
	MONITOR	3	3	--
	DS-CPL	4	4	--
	VMX	5	5	Must be set everywhere on Intel, Ignore on AMD
	RsvdZ	6	6	--
	EST	7	7	--
	TM2	8	8	--
	SSSE3	9	9	--

Index and Register	Name	Start bit	End bit	Virtualized feature set definition between processors in the same system
	CNXTID	10	10	--
	RsvdZ	11	12	--
	CMPXCHG16B	13	13	--
	xTPR	14	14	--
	RsvdZ	15	22	--
	Population Count	23	23	--
			25	
	XSAVE	26	26	Must match
	OSXSAVE	27	27	Must match
	RsvdZ	28	30	--
	Hypervisor Present	31	31	Must be cleared everywhere
EDX	Feature Information			
	FPU	0	0	Must be set everywhere
	VME	1	1	Must be set everywhere
	DE	2	2	Must be set everywhere
	PSE	3	3	Must be set everywhere
	TSC	4	4	Must be set everywhere
	MSR	5	5	Must be set everywhere
	PAE	6	6	Must be set everywhere
	MCE	7	7	Must be set everywhere
	CMPXCHG8B	8	8	Must be set everywhere
	APIC	9	9	Must be set everywhere
	RsvdZ	10	10	--
	SEP	11	11	Must be set everywhere
	MTRR	12	12	Must be set everywhere
	PGE	13	13	Must be set everywhere
	MCA	14	14	Must be set everywhere
	CMOV	15	15	Must be set everywhere
	PAT	16	16	Must be set everywhere
	PSE-36	17	17	Must be set everywhere
	PSN	18	18	--
	CLFSH	19	19	Must be set everywhere
	RsvdZ	20	20	--
	DS	21	21	--
	ACPI	22	22	--
	MMX	23	23	Must be set everywhere
	FXSR	24	24	Must be set everywhere
	SSE	25	25	Must be set everywhere
	SSE2	26	26	Must be set everywhere
	SS	27	27	--
	HTT	28	28	--
	TM	29	29	--
	RsvdZ	30	30	--
	PBE	31	31	--

Index and Register	Name	Start bit	End bit	Virtualized feature set definition between processors in the same system
0x80000000				
EAX	Highest Extended CPUID Leaf	0	31	Hardware specific. If the underlying hardware is AMD-based, the value must be at least 0x80000008. If the underlying hardware is Intel based, the value must be at least 0x80000001
EBX	Processor vendor string	0	31	--
ECX	Processor vendor string	0	31	--
EDX	Processor vendor string	0	31	--
0x80000001				
EAX	Stepping	0	3	--
	Base Model	4	7	Must match
	Base Family	8	11	Must match
	Processor Type	12	13	Must match
	RsvdZ	14	15	--
	Extended Model	16	19	Must match
	Extended Family	20	27	Must match
	RsvdZ	28	31	--
EBX	Brand ID	0	15	Must match
	RsvdZ	16	27	--
	Package Type	28	31	Must match
ECX	Extended feature flag / feature identifiers			
	LahfSahf	0	0	--
	CmpLegacy	1	1	--
	SVM	2	2	Must be set everywhere on AMD, Ignore on Intel
	ExtApicSpace	3	3	--
	AltMovCr8	4	4	--
	ABM	5	5	--
	SSE4A	6	6	--
	MisAlignSse	7	7	--
	3DNowPrefetch	8	8	--
	OSVW	9	9	--
	RsvdZ	10	10	--
	SSE5	11	11	--
	SKINIT	12	12	--
	WDT	13	13	--
	RsvdZ	14	31	--
EDX	Extended			

Index and Register	Name	Start bit	End bit	Virtualized feature set definition between processors in the same system
	feature flags			
	FPU	0	0	Must be set everywhere
	VME	1	1	Must be set everywhere
	DE	2	2	Must be set everywhere
	PSE	3	3	Must be set everywhere
	TSC	4	4	Must be set everywhere
	MSR	5	5	Must be set everywhere
	PAE	6	6	Must be set everywhere
	MCE	7	7	Must be set everywhere
	CMPXCHG8B	8	8	Must be set everywhere
	APIC	9	9	Must be set everywhere
	RsvdZ	10	10	--
	SysCallSysRet	11	11	Must be set everywhere
	MTRR	12	12	Must be set everywhere
	PGE	13	13	Must be set everywhere
	MCA	14	14	Must be set everywhere
	CMOV	15	15	Must be set everywhere
	PAT	16	16	Must be set everywhere
	PSE36	17	17	Must be set everywhere
	RsvdZ	18	19	--
	Execute disabled / No execute	20	20	Must be set everywhere
	RsvdZ	21	21	--
	MmxExt	22	22	--
	MMX	23	23	Must be set everywhere
	FXSR	24	24	Must be set everywhere
	FFXSR	25	25	--
	Page1GB	26	26	--
	RDTSCP	27	27	--
	RsvdZ	28	28	--
	LM	29	29	Must be set everywhere
	3DNowExt	30	30	--
	3DNow	31	31	--
0x80000005	L1 Cache and TLB identifiers (all registers)			
EAX	L1ITlb2and4MSize	0	7	Must match on AMD, Ignore on Intel
	L1ITlb2and4MAssoc	8	15	Must match on AMD, Ignore on Intel
	L1DTlb2and4MSize	16	23	Must match on AMD, Ignore on Intel
	L1DTlb2and4MAssoc	24	31	Must match on AMD, Ignore on Intel
EBX	L1ITlb4KSize	0	7	Must match on AMD, Ignore on Intel
	L1ITlb4KAssoc	8	15	Must match on AMD, Ignore on Intel

Index and Register	Name	Start bit	End bit	Virtualized feature set definition between processors in the same system
	L1DTlb4KSize	16	23	Must match on AMD, Ignore on Intel
	L1DTlb4KAssoc	24	31	Must match on AMD, Ignore on Intel
ECX	L1DcLineSize	0	7	Must match on AMD, Ignore on Intel
	L1DcLinesPerTag	8	15	Must match on AMD, Ignore on Intel
	L1DcAssoc	16	23	Must match on AMD, Ignore on Intel
	L1DcSize	24	31	Must match on AMD, Ignore on Intel
EDX	L1IcLineSize	0	7	Must match on AMD, Ignore on Intel
	L1IcLinesPerTag	8	15	Must match on AMD, Ignore on Intel
	L1IcAssoc	16	23	Must match on AMD, Ignore on Intel
	L1IcSize	24	31	Must match on AMD, Ignore on Intel
0x80000006	L2 Cache and L2 TLB identifiers			
EAX	L2ITlb2and4MSize	0	11	Must match on AMD, Ignore on Intel
	L2ITlb2and4MAssoc	12	15	Must match on AMD, Ignore on Intel
	L2DTlb2and4MSize	16	27	Must match on AMD, Ignore on Intel
	L2DTlb2and4MAssoc	28	31	Must match on AMD, Ignore on Intel
EBX	L2ITlb4KSize	0	11	Must match on AMD, Ignore on Intel
	L2ITlb4KAssoc	12	15	Must match on AMD, Ignore on Intel
	L2DTlb4KSize	16	27	Must match on AMD, Ignore on Intel
	L2DTlb4KAssoc	28	31	Must match on AMD, Ignore on Intel
ECX	L2 Line Size	0	7	Must match
	L2 Lines per tag	8	11	Must match
	L2 Associativity	12	15	Must match
	L2 cache size in kilobytes	16	31	Must match
EDX	L3LineSize	0	7	Must match on AMD, Ignore on Intel
	L3LinesPerTag	8	11	Must match on AMD, Ignore on Intel
	L3Assoc	12	15	Must match on AMD, Ignore on Intel
	RsvdZ	16	17	Must match on AMD,

Index and Register	Name	Start bit	End bit	Virtualized feature set definition between processors in the same system
				Ignore on Intel
	L3Size	18	31	Must match on AMD, Ignore on Intel
0x80000008				
EAX	PhysAddrSize	0	7	Must be less than or equal to 48
	LinAddrSize	8	15	--
	RsvdZ	16	31	--
EBX	RsvdZ	0	31	--
ECX	NC	0	7	Must match on AMD, Ignore on Intel
	RsvdZ	8	11	--
	ApicIdCoreIdSize	12	15	Must match on AMD, Ignore on Intel
	RsvdZ	16	31	--
EDX	RsvdZ	0	31	--
0x8000000A	SVM revision and feature identification			
EAX	SvmRev	0	7	Must match on AMD, Ignore on Intel
	RsvdZ	8	31	--
EBX	NASID	0	31	Must match on AMD, Ignore on Intel
ECX	RsvdZ	0	31	--
EDX	NP	0	0	--
	LBRVirt	1	1	--
	SVML	2	2	--
	NRIPS	3	3	--
	RsvdZ	4	31	--

29 Appendix F: Architectural MSRs

The table below contains a list of architectural MSRs and how they are handled by the hypervisor. The default behavior for MSRs not listed here is passthrough to hardware for root, and #GP for non-root.

MSR Number	MSR Name	Root	Non-Root
0x010	X64_MSR_TIME_STAMP_COUNTER	Virtual TSC	Virtualize
0x01B	X64_MSR_APIC_BASE	Virtualize	Virtualize
0x0FE	X64_MSR_MTRRCAP	Passthrough	Virtualize
0x174	X64_MSR_SYSENTER_CS	Virtualize	Virtualize
0x175	X64_MSR_SYSENTER_ESP	Virtualize	Virtualize
0x176	X64_MSR_SYSENTER_EIP	Virtualize	Virtualize
0x179	X64_MSR_MCG_CAP	Passthrough	0
0x17A	X64_MSR_MCG_STATUS	Virtualize	Virtualize
0x1D9	X64_MSR_DEBUG_CTL	Virtualize	Virtualize
0x200	X64_MSR_MTRR_PHYSBASE0	Passthrough	Virtualize
0x201	X64_MSR_MTRR_PHYSMASK0	Passthrough	Virtualize
0x202	X64_MSR_MTRR_PHYSBASE1	Passthrough	Virtualize
0x203	X64_MSR_MTRR_PHYSMASK1	Passthrough	Virtualize
0x204	X64_MSR_MTRR_PHYSBASE2	Passthrough	Virtualize
0x205	X64_MSR_MTRR_PHYSMASK2	Passthrough	Virtualize
0x206	X64_MSR_MTRR_PHYSBASE3	Passthrough	Virtualize
0x207	X64_MSR_MTRR_PHYSMASK3	Passthrough	Virtualize
0x208	X64_MSR_MTRR_PHYSBASE4	Passthrough	Virtualize
0x209	X64_MSR_MTRR_PHYSMASK4	Passthrough	Virtualize
0x20A	X64_MSR_MTRR_PHYSBASE5	Passthrough	Virtualize
0x20B	X64_MSR_MTRR_PHYSMASK5	Passthrough	Virtualize
0x20C	X64_MSR_MTRR_PHYSBASE6	Passthrough	Virtualize
0x20D	X64_MSR_MTRR_PHYSMASK6	Passthrough	Virtualize
0x20E	X64_MSR_MTRR_PHYSBASE7	Passthrough	Virtualize
0x20F	X64_MSR_MTRR_PHYSMASK7	Passthrough	Virtualize
0x250	X64_MSRMTRR_FIX64K_00000	Passthrough	Virtualize
0x258	X64_MSR_MTRR_FIX16K_80000	Passthrough	Virtualize
0x259	X64_MSR_MTRR_FIX16K_A0000	Passthrough	Virtualize
0x268	X64_MSR_MTRR_FIX4K_C0000	Passthrough	Virtualize
0x269	X64_MSR_MTRR_FIX4K_C8000	Passthrough	Virtualize

0x26A	X64_MSR_MTRR_FIX4K_D0000	Passthrough	Virtualize
0x26B	X64_MSR_MTRR_FIX4K_D8000	Passthrough	Virtualize
0x26C	X64_MSR_MTRR_FIX4K_E0000	Passthrough	Virtualize
0x26D	X64_MSR_MTRR_FIX4K_E8000	Passthrough	Virtualize
0x26E	X64_MSR_MTRR_FIX4K_F0000	Passthrough	Virtualize
0x26F	X64_MSR_MTRR_FIX4K_F8000	Passthrough	Virtualize
0x277	X64_MSR_CR_PAT	Virtualize	Virtualize
0x2FF	X64_MSR_MTRR_DEF_TYPE	Passthrough	Virtualize
0xC0000080	X64_MSR_EFER	Virtualize	Virtualize
0xC0000081	X64_MSR_STAR	Virtualize	Virtualize
0xC0000082	X64_MSR_LSTAR	Virtualize	Virtualize
0xC0000083	X64_MSR_CSTAR	Virtualize	Virtualize
0xC0000084	X64_MSR_SFMASK	Virtualize	Virtualize
0xC0000100	X64_MSR_FS_BASE	Virtualize	Virtualize
0xC0000101	X64_MSR_GS_BASE	Virtualize	Virtualize
0xC0000102	X64_MSR_KERNEL_GS_BASE	Virtualize	Virtualize

30 Appendix G: Vendor-Specific MSRs

G.1 AMD-specific MSRs

The table below contains a list of AMD-specific MSRs and how they are handled by the hypervisor.
The default behavior for MSRs not listed here is passthrough to hardware for root, and #GP for non-root.

MSR Number	MSR Name	Root	Non-Root
0xC0010010	AMD_MSR_SYSCFG	Passthrough	Reads return 0. Writes ignored.
0xC001001F	AMD_MSR_NB_CFG	Passthrough	Reads return 0. Writes ignored.

G.2 Intel-specific MSRs

The table below contains a list of Intel-specific MSRs and how they are handled by the hypervisor.
The default behavior for MSRs not listed here is passthrough to hardware for root, and #GP for non-root.

MSR Number	MSR Name	Root	Non-Root
0x006	INTEL_MSR_MONITOR_FILTER_SIZE	#GP	#GP
0x017	INTEL_MSR_PLATFORM_ID	Passthrough	Reads return 0. Writes ignored.
0x03A	INTEL_MSR_FEATURE_CONTROL	#GP	#GP
0x079	INTEL_MSR_BIOS_UPDT_TRIG	Virtualize	Reads return 0. Writes ignored.
0x08B	INTEL_MSR_BIOS_SIGN_ID	Passthrough	Reads return 8FFFFFFF indicating the best possible patch is already loaded. Writes ignored.
0x1A0	INTEL_MSR_MISC_ENABLE	Passthrough	Reads passthrough, Writes ignored
0x38D	INTEL_MSR_PERF_CAPABILITIES	#GP	#GP
0x38E	INTEL_MSR_PER_GLOBAL_STATUS	#GP	#GP
0x38F	INTEL_MSR_PERF_GLOBAL_CTRL	#GP	#GP
0x390	INTEL_MSR_PERF_GLOBAL_OVF_CTRL	#GP	#GP
0x3F1	INTEL_MSR_PEBS_ENABLE	#GP	#GP
0x480 – 0x48A	INTEL_MSR_VMX_CAPSx	#GP	#GP
0x600	INTEL_MSR_DS_AREA	#GP	#GP

31 Appendix H: Hypervisor Synthetic MSRs

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 3.6.
0x40000001	HV_X64_MSR_HYPERCALL	AccessHypercallMsrs	R/W	Used to enable the hypercall interface. See section 4.12.
0x40000002	HV_X64_MSR_VP_INDEX	AccessVpIndex	R	Specifies the virtual processor's index. See section 10.2.1.
0x40000003	HV_X64_MSR_RESET	AccessSystemResetMsr	R/W	Used to perform a hypervisor-controlled reboot operation. See section 6.3.5.
0x40000010	HV_X64_MSR_VP_RUNTIME	AccessVpRuntimeMsr	R	Specifies the virtual processor's run time in 100ns units. See section 10.3.2.
0x40000020	HV_X64_MSR_TIME_REF_COUNT	AccessPartitionReference Counter	R	Partition-wide reference counter. See section 15.2.
0x40000021	HV_X64_MSR_REFERENCE_TSC	AccessPartitionReference Counter	R	Partition-wide reference time stamp counter. See section 15.4
0x40000070	HV_X64_MSR_EOI	AccessApicMsrs	W	Fast access to the APIC EOI register. See section 13.2.3.
0x40000071	HV_X64_MSR_ICR	AccessApicMsrs	R/W	Fast access to the APIC ICR high and ICR low registers. See section 13.2.3.
0x40000072	HV_X64_MSR_TPR	AccessApicMsrs	R/W	Fast access to the APIC TPR register (use CR8 in 64-bit mode). See section 13.2.3.
0x40000073	HV_X64_MSR_APIC_ASSIST_PAGE	AccessApicMsrs	R/W	Enables lazy EOI processing. See section 13.3.4.
0x40000080	HV_X64_MSR_SCONTROL	AccessSynicMsrs	R/W	Used to control specific behaviors of the synthetic interrupt controller. See section 14.6.1.
0x40000081	HV_X64_MSR_SVERSION	AccessSynicMsrs	R	Specifies the SynIC version. See section 14.6.2.
0x40000082	HV_X64_MSR_SIEFP	AccessSynicMsrs	R/W	Controls the base address of the synthetic interrupt event flag page. See section 14.6.3.
0x40000083	HV_X64_MSR_SIMP	AccessSynicMsrs	R/W	Controls the base address of the synthetic interrupt parameter page. See section 14.6.4.
0x40000084	HV_X64_MSR_EOM	AccessSynicMsrs	W	Indicates the end of message in the SynIC. See section 14.6.6.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x40000090	HV_X64_MSR_SINT0	AccessSynicMsrs	R/W	Configures synthetic interrupt source 0. See section 14.6.5.
0x40000091	HV_X64_MSR_SINT1	AccessSynicMsrs	R/W	Configures synthetic interrupt source 1. See section 14.6.5.
0x40000092	HV_X64_MSR_SINT2	AccessSynicMsrs	R/W	Configures synthetic interrupt source 2. See section 14.6.5.
0x40000093	HV_X64_MSR_SINT3	AccessSynicMsrs	R/W	Configures synthetic interrupt source 3. See section 14.6.5.
0x40000094	HV_X64_MSR_SINT4	AccessSynicMsrs	R/W	Configures synthetic interrupt source 4. See section 14.6.5.
0x40000095	HV_X64_MSR_SINT5	AccessSynicMsrs	R/W	Configures synthetic interrupt source 5. See section 14.6.5.
0x40000096	HV_X64_MSR_SINT6	AccessSynicMsrs	R/W	Configures synthetic interrupt source 6. See section 14.6.5.
0x40000097	HV_X64_MSR_SINT7	AccessSynicMsrs	R/W	Configures synthetic interrupt source 7. See section 14.6.5.
0x40000098	HV_X64_MSR_SINT8	AccessSynicMsrs	R/W	Configures synthetic interrupt source 8. See section 14.6.5.
0x40000099	HV_X64_MSR_SINT9	AccessSynicMsrs	R/W	Configures synthetic interrupt source 9. See section 14.6.5.
0x4000009A	HV_X64_MSR_SINT10	AccessSynicMsrs	R/W	Configures synthetic interrupt source 10. See section 14.6.5.
0x4000009B	HV_X64_MSR_SINT11	AccessSynicMsrs	R/W	Configures synthetic interrupt source 11. See section 14.6.5.
0x4000009C	HV_X64_MSR_SINT12	AccessSynicMsrs	R/W	Configures synthetic interrupt source 12. See section 14.6.5.
0x4000009D	HV_X64_MSR_SINT13	AccessSynicMsrs	R/W	Configures synthetic interrupt source 13. See section 14.6.5.
0x4000009E	HV_X64_MSR_SINT14	AccessSynicMsrs	R/W	Configures synthetic interrupt source 14. See section 14.6.5.
0x4000009F	HV_X64_MSR_SINT15	AccessSynicMsrs	R/W	Configures synthetic interrupt source 15. See section 14.6.5.
0x400000B0	HV_X64_MSR_STIMER0_CONFIG	AccessSyntheticTimerMsrs	R/W	Configuration register for synthetic timer 0. See section 15.3.1.
0x400000B1	HV_X64_MSR_STIMER0_COUNT	AccessSyntheticTimerMsrs	R/W	Expiration time or period for synthetic timer 0. See section 15.3.2.
0x400000B2	HV_X64_MSR_STIMER1_CONFIG	AccessSyntheticTimerMsrs	R/W	Configuration register for synthetic timer 1. See section 15.3.1.
0x400000B3	HV_X64_MSR_STIMER1_COUNT	AccessSyntheticTimerMsrs	R/W	Expiration time or period for synthetic timer 1. See section 15.3.2.
0x400000B4	HV_X64_MSR_STIMER2_CONFIG	AccessSyntheticTimerMsrs	R/W	Configuration register for synthetic timer 2. See section 15.3.1.

MSR Number	MSR Name	Privilege Required (if any)	Access	Description
0x400000B5	HV_X64_MSR_STIMER2_COUNT	AccessSyntheticTimerMsrs	R/W	Expiration time or period for synthetic timer 2. See section 15.3.2.
0x400000B6	HV_X64_MSR_STIMER3_CONFIG	AccessSyntheticTimerMsrs	R/W	Configuration register for synthetic timer 3. See section 15.3.1.
0x400000B7	HV_X64_MSR_STIMER3_COUNT	AccessSyntheticTimerMsrs	R/W	Expiration time or period for synthetic timer 3. See section 15.3.2.
0x400000C1	HV_X64_MSR_POWER_STATE_TRIGGER_C1	CpuManagement	R	Trigger the transition to power state C1
0x400000C2	HV_X64_MSR_POWER_STATE_TRIGGER_C2	CpuManagement	R	Trigger the transition to power state C2
0x400000C3	HV_X64_MSR_POWER_STATE_TRIGGER_C3	CpuManagement	R	Trigger the transition to power state C3
0x400000D1	HV_X64_MSR_POWER_STATE_CONFIG_C1	CpuManagement	R/W	Configure the recipe for power state transitions to C1
0x400000D2	HV_X64_MSR_POWER_STATE_CONFIG_C2	CpuManagement	R/W	Configure the recipe for power state transitions to C2
0x400000D3	HV_X64_MSR_POWER_STATE_CONFIG_C3	CpuManagement	R/W	Configure the recipe for power state transitions to C3
0x400000E0	HV_X64_MSR_STATS_PARTITION_RETAIL_PAGE	AccessStatsMsr	R/W	Map the guest's retail partition statistics page
0x400000E1	HV_X64_MSR_STATS_PARTITION_INTERNAL_PAGE	AccessStatsMsr	R/W	Map the guest's internal partition statistics page
0x400000E2	HV_X64_MSR_STATS_VP_RETAIL_PAGE	AccessStatsMsr	R/W	Map the guest's retail VP statistics page
0x400000E3	HV_X64_MSR_STATS_VP_INTERNAL_PAGE	AccessStatsMsr	R/W	Map the guest's internal VP statistics page
0x400000F0	HV_X64_MSR_GUEST_IDLE	AccessGuestIdleMsr	R	Trigger the guest's transition to the idle power state

32 Appendix I: Event Log Entries

Note: In the table below, rows with bold entries in the “Event Name” column and 8-byte entries in the “Event Number” column represent event groups, not individual events. Not all groups have events defined, but the groups are included for completeness.

Event Name	Event Number	Event Description	Parameters
Boot Manager	0x0000010000000000		
Dispatch Manager	0x0000000000000200		
Hypercall	0x0000000000000400		
HV_TR_HC_HYPERCALL	0x1A	Hypercall	Guest RIP, Input Value
Intercept Manager	0x0000000000000800		
HV_TR_IM_GUEST_EXCEPTION	0x20	Guest exception	Guest RIP, Type
HV_TR_IM_MSR_READ	0x21	MSR Read	Guest RIP, Index, Value (0 if unsuccessful)
HV_TR_IM_MSR_WRITE	0x22	MSR Write	Guest RIP, Index, Value
HV_TR_IM_ CR_READ	0x23	Control Register Read	Guest RIP, Index, Value (0 if unsuccessful)
HV_TR_IM_CR_WRITE	0x24	Control Register Write	Guest RIP, Index, Value
HV_TR_IM_HLT_INSTRUCTION	0x25	Hlt Instruction	Guest RIP
HV_TR_IM_MWAIT_INSTRUCTION	0x26	MWAIT Instruction	Guest RIP, MWAIT Extension, MWAIT Hint
HV_TR_IM_CPUID_INSTRUCTION	0x27	CPUID Instruction	Guest RIP, EAX Input, ECX Input
HV_TR_IM_IO_PORT_READ	0x28	IO Port Read	Guest RIP, Port, Value (0 if unsuccessful)
HV_TR_IM_IO_PORT_WRITE	0x29	IO Port Write	Guest RIP, Port, Value
HV_TR_IM_EXTERNAL_INTERR	0x2A	External	Vector

Event Name	Event Number	Event Description	Parameters
UPT		interrupt	
HV_TR_IM_ INTERRUPT_PENDING	0x2B	Virtual Interrupt Pending	Guest RIP
HV_TR_IM_GUEST_SHUTDOWN	0x2C	Guest Shutdown	Guest RIP
HV_TR_IM_EMULATED_INSTRUCTION	0x2D	Emulated Instruction	Guest RIP
HV_TR_IM_NMI_INTERRUPT	0x2E	Non-Maskable Interrupt	Guest RIP
HV_TR_IM_INVLPG_INSTRUCTION	0x2F	INVLPG Instruction	Guest RIP, Guest VA
HV_TR_IM_IRET_INSTRUCTION	0x30	IRET Instruction	Guest RIP
HV_TR_IM_TASK_SWITCH	0x31	Task Switch	Guest RIP
HV_TR_IM_INVD_INSTRUCTION	0x32	INVD Instruction	Guest RIP
HV_TR_IM_ DR_ACCESS	0x33	Debug Register Access	Guest RIP
HV_TR_IM_FERR_FREEZE	0x34	FERR Freeze	Guest RIP
HV_TR_IM_REAL_MODE_INTERRUPT	0x35	Emulated Real-Mode Interrupt	Guest RIP
HV_TR_IM_MEMORY_INTERCEPT	0x36	Memory Intercept	Guest RIP, GPA Intercept Type
HV_TR_IM_REFLECTED EXCEPTION	0x37	Reflected Guest Exception	Guest RIP, Exception Type
Instruction Completion	0x0000000000001000		
Object Manager	0x0000000000002000		
HV_TR_OB_CREATE_PARTITION	0x40	Create Partition	Partition ID, Hv Status
HV_TR_OB_DELETE_PARTITION	0x41	Delete Partition	Partition ID, Hv Status
Partition Manager	0x0000000000004000		
HV_TR_PT_REFERENCE_TIME	0x46	Partition Reference Time	TscOffset, ReferenceTime Offset
Virtual Processor Manager	0x0000000000008000		
HV_TR_VP_CREATE_VP	0x4B	Create Virtual Processor	Partition ID, Thread ID,

Event Name	Event Number	Event Description	Parameters
			Hv Status
HV_TR_VP_DELETE_VP	0x4C	Delete Virtual Processor	Partition ID, Thread ID, Hv Status
Synthetic Interrupt Controller	0x0000000000010000		
Synthetic Timers	0x0000000000020000		
Address Manager – Guest Virtual Addresses	0x0000000000040000		
HV_TR_AM_GVA_GROW_VIRTUAL_TLB	0x70	Grow Virtual TLB	Old Count, New Count
HV_TR_AM_GVA_SHRINK_VIRTUAL_TLB	0x71	Shrink Virtual TLB	Old Count, New Count, Free Count
HV_TR_AM_GVA_FLUSH_VIRTUAL_TLB	0x72	FlushVirtual TLB	Partition ID
Address Manager	0x0000000000080000		
Virtual Abstraction Layer	0x0000000000100000		
Virtualization Manager	0x0000000000200000		
Scheduler	0x0000000000400000		
HV_TR_SCH_CONTEXT_SWITCH	0xB0	Context Switch	Partition ID, Thread ID, TSC Offset (perf only)
Threads	0x0000000000800000		
Timers	0x0000000001000000		
Kernel	0x0000000002000000		
Memory Manager	0x0000000004000000		

Event Name	Event Number	Event Description	Parameters
Profiler	0x0000000008000000		
HV_TR_SCH_PROFILER_SAMPLE	0xF0	Context Switch	Partition ID, Thread ID, TSC Offset (perf only)
HV_TR_PROFILER_HV_MODULE	0Xf1		

33 Appendix J: Statistics Counter Definitions

The hypervisor exposes statistics counters to guests, upon request, using overlay pages in the partition's GPA space. Each page is a packed collection of statistics groups. Each group is prefixed by a header that contains the group's identification, the format version and the size of the group in bytes (not including the header). The array of values within the group (after the header) may be accessed using a zero-based index into the array. Index zero references the group's header. Each 64-bit value is of a particular type.

This appendix describes each supported statistics page and the groups that it may contain. Statistics Counters are discussed in detail in chapter 21.

J.1 Hypervisor Counters

Hypervisor Counters describe hypervisor-wide states and items. There is only one Hypervisor Counter Object in the system consisting of only architectural values. As counters of a global class they can only be accessed by the root partition.

J.1.1 Architected Hypervisor Counters Group

Group identifier: HV_STATISTICS_TYPE_HCA_ID
Group version: HV_STATISTICS_TYPE_HCA_VERSION

Architected Hypervisor Counters (accessible from the root partition only)			
Index	Counter	Type	Description
1	Logical Processors	UINT64	The number of logical processors present in the system.
2	Partitions	UINT64	The number of partitions (virtual machines) present in the system.
3	Total Pages	UINT64	The number of bootstrap and deposited pages in the hypervisor.
4	Virtual Processors	UINT64	The number of virtual processors present in the system.
5	Monitored Notifications	UINT64	The number of monitored notifications registered with the hypervisor.

J.2 Logical Processor Counters

Counters for Logical Processor Objects provide information about logical processor-related states and items. The number of Logical Processor Objects matches the number of logical processors in the system and consists of both architectural and release-specific values. As counters of a global class they can be only accessed by the root partition.

J.2.1 Architected Logical Processor Counters Group

Group identifier: HV_STATISTICS_TYPE_LPA_ID
Group version: HV_STATISTICS_TYPE_LPA_VERSION

Architected Logical Processor Counters (accessible from the root partition only)			
Index	Counter	Type	Description
1	Global Time	UINT64	The global time on this logical processor.
2	Total Run Time	UINT64	The total time (in 100ns) spent by the processor in guest and hypervisor code.
3	Hypervisor Run Time	UINT64	The total time (in 100ns) spent by the processor in hypervisor code.
4	Hardware Interrupts/sec	UINT64	The rate of hardware interrupts on the processor (excluding hypervisor interrupts).
5	Context Switches/sec	UINT64	The rate of virtual processor context switches on the processor.
6	Inter-Processor Interrupts/sec	UINT64	The rate of hypervisor inter-processor interrupts delivered to the processor.
7	Scheduler Interrupts/sec	UINT64	The rate of hypervisor scheduler interrupts on the processor.
8	Timer Interrupts/sec	UINT64	The rate of hypervisor timer interrupts on the processor.
9	Inter-Processor Interrupts Sent/sec	UINT64	The rate of hypervisor inter-processor interrupts sent by the processor.
10	Processor Halts/sec	UINT64	The rate of entries into a halt state by the processor.
11	Monitor Transition Cost	UINT64	The hardware cost of transitions into the hypervisor.
12	Context Switch Time	UINT64	The total time (in nanoseconds) spent switching between virtual processors.
13	C1 Transitions/sec	UINT64	The rate at which that CPU enters the C1 low-power idle state.
14	% C1 Time	UINT64	The percentage of time the processor spends in the C1 low-power idle state. % C1 Time is a subset of the total processor idle time.
15	C2 Transitions/sec	UINT64	The rate at which that CPU enters the C2 low-power idle state.
16	% C2 Time	UINT64	The percentage of time the processor spends in the C2 low-power idle state. % C2 Time is a subset of the total processor idle time.
17	C3 Transitions/sec	UINT64	The rate at which that CPU enters the C3 low-power idle state.
18	% C3 Time		The percentage of time the processor spends in the C3 low-power idle state. % C3 Time is a subset of the total processor idle time.

J.3 Partition Counters

Counters for Partition Objects provide information about states and items for a particular partition. The number of Partition Objects matches the number of partitions present in the system consisting of only architectural values. As counters of a local class they can be accessed by a partition and its parent. In Version 1, the root partition is the parent partition.

J.3.1 Architected Partition Counters Group

Group identifier: HV_STATISTICS_TYPE_PA_ID
Group version: HV_STATISTICS_TYPE_PA_VERSION

Architected Partition Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
1	Virtual Processors	UINT64	The number of virtual processors present in the partition.

J.3.2 Release-specific Partition Counters Group

Group identifier: HV_STATISTICS_TYPE_PV_ID
Group version: HV_STATISTICS_TYPE_PV_VERSION

Release-specific Partition Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
1	Virtual TLB Pages	UINT64	The number of pages used by the virtual TLB of the partition.
2	Address Spaces	UINT64	The number of address spaces in the virtual TLB of the partition.
3	Deposited Pages	UINT64	The number of pages deposited into the partition.
4	GPA Pages	UINT64	The number of pages present in the GPA space of the partition (zero for root partition).
5	GPA Space Modifications/sec	UINT64	The rate of modifications to the GPA space of the partition.
6	Virtual TLB Flush Entries/sec	UINT64	The rate of flushes of the entire virtual TLB.
7	Recommended Virtual TLB Size	UINT64	The recommended number of pages to be deposited for the virtual TLB.

J.4 Virtual Processor Counters

Counters for Virtual Processor Objects provide information about states and items for a particular virtual processor. The number of Virtual Processor Objects matches the number of virtual processors present in a given partition and consists of both architectural and release-specific values. As counters of a local class they can be accessed by the partition and its parent. In Version 1, the root partition is the parent partition.

J.4.1 Architected Virtual Processor Counters Group

Group identifier: HV_STATISTICS_TYPE_VPA_ID
Group version: HV_STATISTICS_TYPE_VPA_VERSION

Architected Virtual Processor Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
1	Total Run Time	UINT64	The total run time (in 100ns) spent by the processor in guest and hypervisor code.
2	Hypervisor Run Time	UINT64	The total time (in 100ns) spent by the virtual processor in hypervisor code.

J.4.2 Release-specific Virtual Processor Counters Group

Group Identifier: HV_STATISTICS_TYPE_VPV_ID
Group version: HV_STATISTICS_TYPE_VPV_VERSION

Release-specific Virtual Processor Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
1	Hypercalls/sec	UINT64	The rate of the hypercalls made by guest code on the virtual processor.
2	Hypercalls Cost	UINT64	The average time (in nanoseconds) spent processing a hypercall.
3	Page Invalidations/sec	UINT64	The rate of INVLPG instructions executed by guest code on the virtual processor.
4	Page Invalidations Cost	UINT64	The average time (in nanoseconds) spent processing an INVLPG instruction.
5	Control Register Accesses/sec	UINT64	The rate of control register accesses by guest code on the virtual processor.
6	Control Register Accesses Cost	UINT64	The average time (in nanoseconds) spent processing a control register access.
7	IO Instructions/sec	UINT64	The rate of IO instructions executed by guest code on a virtual processor.
8	IO Instructions Cost	UINT64	The average time (in nanoseconds) spent processing an IO instruction.
9	HLT Instructions/sec	UINT64	The rate of HLT instructions executed by guest code on a virtual processor.
10	HLT Instructions Cost	UINT64	The average time (in nanoseconds) spent processing an HLT instruction.
11	MWAIT Instructions/sec	UINT64	The rate of MWAIT instructions executed by guest code on a virtual processor.

Release-specific Virtual Processor Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
12	MWAIT Instructions Cost	UINT64	The average time (in nanoseconds) spent processing an MWAIT instruction.
13	CPUID Instructions/sec	UINT64	The rate of CPUID instructions executed by guest code on a virtual processor.
14	CPUID Instructions Cost	UINT64	The average time (in nanoseconds) spent processing an CPUID instruction.
15	MSR Accesses/sec	UINT64	The rate of MSR instructions executed by guest code on a virtual processor.
16	MSR Accesses Cost	UINT64	The average time (in nanoseconds) spent processing an MSR instruction.
17	Other Intercepts/sec	UINT64	The rate of other intercepts triggered by guest code on a virtual processor.
18	Other Intercepts Cost	UINT64	The average time (in nanoseconds) spent processing other intercepts.
19	External Interrupts/sec	UINT64	The rate of external interrupts received by the hypervisor while executing guest code on the virtual processor.
20	External Interrupts Cost	UINT64	The average time (in nanoseconds) spent processing an external interrupt.
21	Pending Interrupts/sec	UINT64	The rate of intercepts due to a task priority (TPR) reduction by guest code on the virtual processor.
22	Pending Interrupts Cost	UINT64	The average time (in nanoseconds) spent processing a pending interrupt intercept.
23	Emulated Instructions/sec	UINT64	The rate of emulated instructions while executing guest code on the virtual processor.
24	Emulated Instructions Cost	UINT64	The average time (in nanoseconds) spent emulating an instruction.
25	Debug Register Accesses/sec	UINT64	The rate of debug register accesses by guest code on the virtual processor.
26	Debug Register Accesses Cost	UINT64	The average time (in nanoseconds) spent handling a debug register access.
27	Page Fault Intercepts/sec	UINT64	The rate of page fault exceptions intercepted by the hypervisor while executing guest code on the virtual processor.

Release-specific Virtual Processor Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
28	Page Fault Intercepts Cost	UINT64	The average time (in nanoseconds) spent processing a page fault intercept.
29	Guest Page Table Maps/sec	UINT64	The rate of map operations for guest page table pages.
30	Large Page TLB Fills/sec	UINT64	The rate of virtual TLB misses on large pages.
31	Small Page TLB Fills/sec	UINT64	The rate of virtual TLB misses on 4K pages.
32	Reflected Guest Page Faults/sec	UINT64	The rate of page fault exceptions delivered to the guest.
33	APIC MMIO Accesses/sec	UINT64	The rate of APIC MMIO register accesses by guest code on the virtual processor.
34	IO Intercept Messages/sec	UINT64	The rate of IO intercept messages sent to the parent partition.
35	Memory Intercept Messages/sec	UINT64	The rate of memory intercept messages sent to the parent partition.
36	APIC EOI Accesses/sec	UINT64	The rate of APIC EOI register writes by guest code on the virtual processor.
37	Other Messages/sec	UINT64	The rate of other intercept messages sent to the parent partition.
38	Page Table Allocations/sec	UINT64	The rate of page table allocations in the virtual TLB.
39	Logical Processor Migrations/sec	UINT64	The rate of migrations by the virtual processor to a different logical processor.
40	Address Space Evictions/sec	UINT64	The rate of address space evictions in the virtual TLB.
41	Address Space Switches/sec	UINT64	The rate of address space switches by guest code on the virtual processor.
42	Address Domain Flushes/sec	UINT64	The rate of explicit flushes of the virtual TLB by guest code on the virtual processor.
43	Address Space Flushes/sec	UINT64	The rate of explicit flushes of one address space by guest code on the virtual processor.
44	Global GVA Range Flushes/sec	UINT64	The rate of explicit flushes of a virtual address range in all address spaces by guest code on the virtual processor.
45	Local Flushed GVA Ranges/sec	UINT64	The rate of explicit flushes of a virtual address range in one address space by guest code on the virtual processor.
46	Page Table Evictions/sec	UINT64	The rate of page table evictions in the virtual TLB.

Release-specific Virtual Processor Counters (accessible from a partition and its parent)			
Index	Counter	Type	Description
47	Page Table Reclamations/sec	UINT64	The rate of reclamations of unreferenced page tables in the virtual TLB.
48	Page Table Resets/sec	UINT64	The rate of page table resets In the virtual TLB.
49	Page Table Validations/sec	UINT64	The rate of page table validations to remove state entries in the virtual TLB.
50	APIC TPR Accesses/sec	UINT64	The rate of APIC TPR accesses by guest code on the virtual processor.
51	Page Table Write Intercepts/sec	UINT64	The rate of write intercepts on guest page tables by guest code on the virtual processor.
52	Synthetic Interrupts/sec	UINT64	The rate of synthetic interrupts delivered to the virtual processor.
53	Virtual Interrupts/sec	UINT64	The rate of interrupts (including synthetic interrupts) delivered to the virtual processor.
54	APIC IPIs Sent/sec	UINT64	The rate of APIC inter-processor interrupts (including to self) sent to the virtual processor.
55	APIC Self IPIs Sent/sec	UINT64	The rate of APIC interrupts sent by the virtual processor to itself.
56	GPA Space Hypercalls/sec	UINT64	The rate of Guest Physical Address Space hypercalls made by guest code on the virtual processor.
57	Logical Processor Hypercalls/sec	UINT64	The rate of Logical Processor hypercalls made by guest code on the virtual processor.
58	Long Spin Wait Hypercalls/sec	UINT64	The rate of Long Spin Wait hypercalls made by guest code on the virtual processor.
59	Other Hypercalls/sec	UINT64	The rate of other hypercalls made by guest code on the virtual processor.
60	Synthetic Interrupt Hypercalls/sec	UINT64	The rate of Synthetic Interrupt hypercalls made by guest code on the virtual processor.
61	Virtual Interrupt Hypercalls/sec	UINT64	The rate of Virtual Interrupt hypercalls made by guest code on the virtual processor.
62	Virtual MMU Hypercalls/sec	UINT64	The rate of Virtual MMU hypercalls made by guest code on the virtual processor.
63	Virtual Processor Hypercalls/sec	UINT64	The rate of Virtual Processor hypercalls made by guest code on the virtual processor.

Index

A

C

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W