UltraSPARC Virtual Machine Specification

(The sun4v architecture and Hypervisor API specification)

Revision 1.0

Please send comments and queries to:
hypervisor@sun.com

Table of Contents

1 Introduction

Sun's UltraSPARC T1 processor has been designed to incorporate hypervisor technology in order to present a virtualized machine environment to any guest operating system running upon it. The resulting software model for a guest operating system is referred to as the "sun4v" architecture. This virtual machine environment is implemented with a thin layer of firmware software (the "UltraSPARC Hypervisor") coupled with hardware extensions providing protection. The UltraSPARC Hypervisor not only provides system services required by the operating system, but it also enables the separation of platform resources into self-contained partitions (logical domains) each capable of supporting an independent operating system image.

This document details the virtual machine environment and the calling conventions of the APIs provided to a sun4v domain by the underlying UltraSPARC hypervisor. The intended audience for this document is operating system and firmware engineers porting to the sun4v architecture.

The API serves two principal purposes:

1. To enable the supervisor to request services and operations to be performed on its behalf by the hypervisor.
2. To inform the hypervisor of information it expects from the supervisor, for example the size and location of the interrupt delivery queues.

1.1 Related specifications

The sun4v architecture provides a virtual machine environment through a conjunction of platform hardware and hypervisor software. This virtual machine environment consists of a combination of machine registers described by a programmer's reference manual, and a set of software services provided via the hypervisor APIs described in this document.

The hardware registers available within a virtual machine environment, (described in the UltraSPARC Architecture 2005 manual), form the basis of the sun4v hardware architecture. This architecture incorporates the Level-1 SPARC v9 specification. However, it supersedes and extends the Level-2 SPARC v9 specification in describing the programming model, register and exception interfaces for privileged mode software.

In addition to the UltraSPARC Architecture 2005 manual, processor specific details for the UltraSPARC T1 processor are provided in the "UltraSPARC T1™ Supplement" manual.

At the time of writing the latest versions of this specification, the UltraSPARC Architecture 2005 manual and the UltraSPARC T1™ Supplement are available from the OpenSPARC website (http://www.opensparc.org). The reader is recommended to visit the OpenSPARC website on a regular basis for the most recent versions of these specifications.

The names "Niagara" and "Niagara-1" refer to the UltraSPARC T1 processor.

2 Hypervisor call conventions

Hypervisor API calls are made through the use of a trap (`Tcc`) instruction using *sw_trap_numbers* 0x80 and above. The calling convention has two forms; fast-trap and hyper-fast-trap. The principle difference between these two forms is whether the function number is passed in a register or is encoded in the trap instruction itself. The latter is the faster form, but has a limited number of possible functions, and is therefore reserved for performance critical operations only.

2.1 Hyper-fast traps

This trap mechanism encodes the API function number (0x80 + a 7bit value) in the Tcc instruction's *sw_trap_number* itself, and therefore provides the fastest possible method of reaching the actual function implementation. The calling convention is as follows:

Register	Input	Output
%o0	argument 0	return status
%o1	argument 1	return value1
%o2	argument 2	return value2
%o3	argument 3	return value3
%o4	argument 4	return value4

All arguments and return values are 64-bits unless explicitly stated by the description of a specific API service. Further arguments may be passed in memory, as defined on a per function basis.

2.2 Fast traps

Fast traps are the preferred mechanism for hypervisor API calls. Fast trap API calls primarily use *sw_trap_number* 0x80 in the Tcc instruction, with the required function number provided as a 64bit value in register %o5. The calling convention is as follows:

Register	Input	Output
%o5	function number	undefined
%o0	argument 0	return status
%o1	argument 1	return value 1
%o2	argument 2	return value 2
%o3	argument 3	return value 3
%o4	argument 4	return value 4

All arguments and return values are 64-bits **unless** explicitly stated by the description of a specific API service. Further arguments may be passed in memory, as defined on a per function call basis.

2.3 Post hypervisor trap processing

The following convention is used, unless explicitly described for a particular API service:

- All API services resume executing at the next logical instruction after the service trap as with a *done* instruction.
- All sun4v defined registers are preserved across an API service except as explicitly stated below;

- Registers providing arguments to an API service (including the function number %o5 for fast traps) should be considered volatile, and their values upon return are undefined unless they are explicitly specified on a per-service basis. Registers not used for passing arguments or returning values are preserved across the API service.
- Upon return from the API service, the returned status is given in register %o0. A value of zero in %o0 indicates successful execution of the API service, all other values indicate an error status (as defined in section 25.5).

- If an invalid *sw_trap_number* is issued, or if an invalid function number is specified, the hypervisor will return with EBADTRAP (as defined in section 25.5) in %o0.
- All 64 bits of the argument or return values are significant.

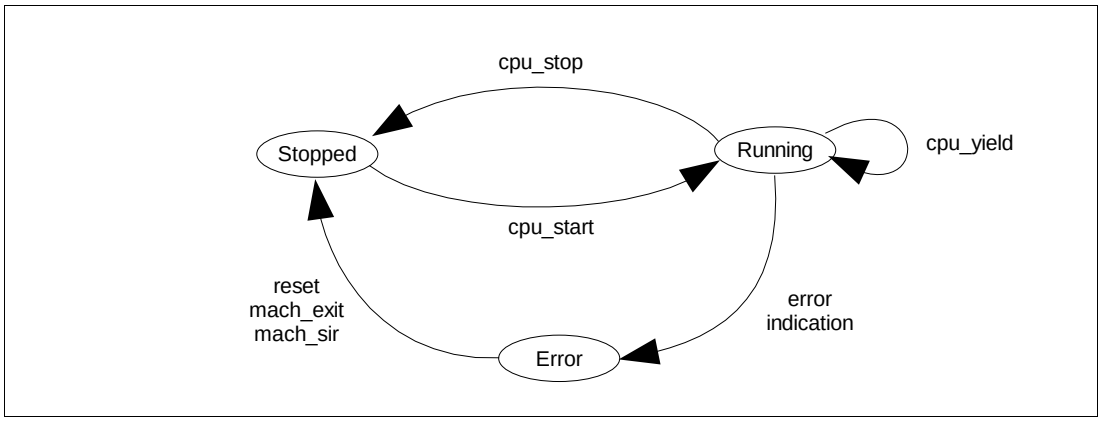
3 State definitions

3.1 Guest states

Each virtual CPU can have one of three different states:

Stopped	CPU is stopped, not executing code, and may be started via the cpu_start API service
Running	CPU is executing
Error	CPU is in error, and no longer executing code

The relationship of these CPU states and hypervisor services may be summarized with the state diagram below:



3.2 Initial guest environment

The initial state of each sun4v virtual CPU is defined in the Sun4v Architecture Specification. Initial register state is duplicated here together with initial register configuration performed by the hypervisor for completeness.

3.3 Privileged registers

Register(s)	Initial Value
%cwp	0
%cansave	NWIN-2
%cleanwin	NWIN-2
%canrestore	0
%otherwin	0
%wstate	0
%pstate	all 0 except pstate.priv=1, pstate.mm=tso
%tl	MAXPTL (2)
%gl	MAXPGL (2)
%pil	MAXPIL (0xf)
%tba	current rtba
%tt	POR

3.3.1 Non-Privileged Registers

Register(s)	Initial Value
%g1-%g7	0
%i0[%cwp]	real address of startup memory segment
%i1[%cwp]	size of startup memory segment
%i2-%i7[%cwp]	0
%i0-%i7[all other windows]	0
%l0-%l7[all windows]	0
%d0-%d62	Binary 0
%fsr	0

3.3.2 Ancillary State Registers

Register(s)	Initial Value
asr0 (%y)	0
asr2 (%ccr)	0
asr3 (%asi)	ASI_REAL
asr4 (%tick)	>0, npt=0
asr5 (%pc)	current pc
asr6 (%fprs)	0
asr19 (%gsr)	0
asr22 (%softint)	0
asr24 (%stick)	>0, npt=0
asr25 (%stick_cmpr)	0 with interrupts disabled (bit 63=1)

3.3.3 Internal memory-mapped registers

Register(s)	Initial Value
ASI_SCRATCHPAD, VA=0x00	0
ASI_SCRATCHPAD, VA=0x08	0
ASI_SCRATCHPAD, VA=0x10	0
ASI_SCRATCHPAD, VA=0x18	0
ASI_SCRATCHPAD, VA=0x20	0 if implemented
ASI_SCRATCHPAD, VA=0x28	0 if implemented
ASI_SCRATCHPAD, VA=0x30	0
ASI_SCRATCHPAD, VA=0x38	0
ASI_MMU, VA=0x08 (primary ctx)	0
ASI_MMU, VA=0x10 (secondary ctx)	0
ASI_MMU, VA=0xn08 (for valid {n} > 0)	0
ASI_MMU, VA=0xn10 (for valid {n} > 0)	0
ASI_QUEUE, VA=0x3c0 (cpu mondo head)	0
ASI_QUEUE, VA=0x3c8 (cpu mondo tail)	0
ASI_QUEUE, VA=0x3d0 (dev mondo head)	0

Register(s)	Initial Value
ASI_QUEUE, VA=0x3d8 (dev mondo tail)	0
ASI_QUEUE, VA=0x3e0 (res. error head)	0
ASI_QUEUE, VA=0x3e8 (res. error tail)	0
ASI_QUEUE, VA=0x3f0 (nres. error head)	0
ASI_QUEUE, VA=0x3f8 (nres. error tail)	0

3.3.4 CPU-specific Registers

Platform specific performance counters will be configured such that exceptions/interrupts are disabled.

3.4 Other initial guest state

MMU state is disabled.

MMU fault status area location is undefined.

TSB info is undefined.

All queue base addresses and sizes are undefined.

One CPU is placed into the running state, all other CPUs are in the stopped state.

Initial guest soft state is set to SS_TRANSITION, with an empty (NUL) description string.

4 Addressing Models

4.1 Background

This section defines the sun4v memory management architecture. The intent is to provide a memory addressing capability for a virtualized architecture at the same time removing the explicit dependence on hardware mechanisms for virtual memory management. Mechanisms are provided to privileged mode to manipulate the memory made available, and in turn to virtualize and make that memory available to non-privileged mode processes.

4.2 Address types

The sun4v architecture has two address types, as in legacy architectures. The main difference is that *virtual adresses* are translated to *real addresses,* as opposed to being translated to *physical addresses.* This change is made in order to enable the segregation of physical memory into multiple partitions.

Virtual addresses are translated by an MMU in order to locate data in physical memory. This definition is unchanged from current systems for nonprivileged and privileged mode addresses.

Real addresses are provided to privileged mode code to describe the underlying physical memory allocated to it. Translation storage buffers (TSBs) maintained by privileged mode code are used to translate privileged or nonprivileged mode virtual addresses into real addresses. MMU bypass addresses in privileged mode are also real addresses.

4.3 Address spaces

Address spaces are unchanged from UltraSPARC-1. Primary and secondary virtual addresses are associated with context identifiers that are used by privileged code to create multiple address spaces.

4.4 Address space identifiers

Instructions can explicitly specify an address space via address space identifiers. All the SPARC v9 ASI definitions are unchanged for sun4v, and a number of new ASIs are also defined. ASIs related to memory management are described below:

ASI #	ASI Name
0x14	REAL_MEM
0x15	REAL_IO
0x1c	REAL_MEM_LITTLE
0x1d	REAL_IO_LITTLE
0x21	MMU

4.4.1 ASI 0x14 & 0x1c : REAL_MEM{_LITTLE}

This ASI provides privileged mode access to cached memory using a real rather than virtual address. For this access the context id is unused. A *nonresumable_error* trap occurs if the access cannot be completed.

4.4.2 ASI 0x15 & 0x1d : REAL_IO{_LITTLE}

This ASI provides privileged mode access to uncached memory addresses using a real rather than virtual address. For this access the context id is unused. A *nonresumable_error* trap occurs if the access cannot be completed.

4.4.3 ASI 0x26 & 0x2E : REAL_QUAD{_LITTLE}

This ASI provides atomic access to 16 bytes of data using real addresses. A *mem_address_not_aligned* trap is taken if the address is not 16 byte aligned.

4.4.4 ASI 0x21 : MMU

The sun4v MMU interface consists of the following registers:

Register	Address
PRIMARY_CONTEXTn	0xn08
SECONDARY_CONTEXTn	0xn10

These registers are used for the primary and secondary context values utilized by the processor TLB for distinguishing address space contexts. The number of primary and secondary context registers provided is implementation dependent subject to the following rules:

1. The number of primary context registers must be the same as the number of secondary context registers.
2. The context registers must start with n=0, and be arranged sequentially without gaps. So, for example with 4 registers, n=0,1,2,3.
3. The number of bits provided must be the same for all context registers.
4. For ease of programming, a write to PRIMARY_CONTEXT0 causes the same context value to be written to all other PRIMARY_CONTEXT registers. Similarly, a write to SECONDARY_CONTEXT0 causes the same context value to be written to all other SECONDARY_CONTEXT registers.

Sun4v provides a minimum of 13 bits of context (bits 0 through 12). Further bits (from 13 and up) may be provided as an implementation dependent feature. The maximum number of bits for a given hardware platform are given as a property in the guest's machine description. Privileged code is responsible for honoring the number of bits supported by hardware.

4.4.4.1 Programming note

The policy of how privileged code chooses to use the primary and secondary context registers is beyond the scope of this document. However, because sun4v only guarantees the existence of PRIMARY_CONTEXT0 and SECONDARY_CONTEXT0 it is recommended that these be used as process private context registers, while any remaining context registers be used for possibly shared context address spaces.

4.4.4.2 Translation conflicts

For sun4v platforms that implement more than one primary and more than one secondary context register privileged code must ensure that no more than one page translation is allowed to match at any time.

An illustration of erroneous behavior is as follows: an operating system constructs a mapping for virtual address A valid for context P, it then constructs a mapping for address A for context Q. By setting PRIMARY_CONTEXT0 to P and PRIMARY_CONTEXT1 to Q both mappings would be active simultaneously - potentially with conflicting translations for address A.

Care must be taken not to construct such scenarios.

To prevent errors/data corruption sun4v processors will detect such conflicts, flush the TLB, and issue a *{data/instruction}_access_exception*.

4.4.4.3 Barrier rules

By definition changing either the primary or secondary context registers has side effects on processor behavior. The following table describes the behavior of a `stxa` to these registers.

	@ TL = 0	@ TL > 0
PRIMARY_CONTEXT	undefined; privileged code should not change PRIMARY_CONTEXT at TL=0	membar #Sync, DONE or RETRY are required for effects to be guaranteed observable, otherwise results are undefined.
SECONDARY_CONTEXT	membar #Sync is required for effects to be guaranteed observable, otherwise results are undefined	membar #Sync, DONE or RETRY are required for effects to be guaranteed observable, otherwise results are undefined.

4.5 Translation mappings

Privileged code describes virtual to real address mappings to manage its virtual address spaces. These mappings are declared either as translation table entries (TTEs) in a translation storage buffer (TSB) described in section 12.1, or can be established directly by the use of the hypervisor API call mmu_map_perm_addr (§12.7.7). This call can also be used to establish a limited number of "locked" mappings for which privileged code cannot tolerate an MMU miss trap.

4.6 MMU Demap support

Privileged mode demap operations become hypervisor API calls.

It is important to note that sun4v provides a coherent demap capability for the privileged mode. The demap API call takes a list of virtual CPUs for which the demap operation is to be applied.

The following three demap operations are required for sun4v:

Demap Page The translations demapped match the virtual address and context id designated.

Demap Context the translations demapped match the context id designated.

Demap All this demaps all translations.

4.7 MMU traps

MMU privilege mode traps are a subset of the MMU traps described in the SPARC v9 specification:

{instruction,data}_access_mmu_miss

shall be generated when a nonprivileged or privileged mode access does not have a translation in any of the TSBs.

data_access_protection

shall be generated when a nonprivileged or privileged mode access matches a translation that does not allow the requested action, i.e. store when TTE write enable field is clear. This also enables software simulation of a TLB entry *modified* bit, as well as fast *copy-on-write* page processing.

To speed processing of a copy-on-write or modified-bit usage, the faulting TLB entry is guaranteed flushed from the local CPU's TLB upon entry of this exception. Thus, in the common case, no flush operation needs to be generated before enabling write permission in the faulting TTE.

{instruction,data}_access_exception

shall be generated as the result of a nonprivileged mode access when TTE privilege field is set, or as the result of an instruction fetch when the TTE execute permission bit is not set, or as the result of two conflicting translation matches for the same virtual address.

fast_{instruction,data}_access_MMU_miss

shall be generated when a nonprivileged or privileged mode access does not have a translation in any TLB and no TSB is specified for the virtual cpu.

fast_data_access_protection

shall be generated when no TSB is specified for the virtual cpu and a nonprivileged or privileged mode access matches a TLB translation that does not allow the requested action, i.e. store when TTE write enable field is clear. This also enables software simulation of a TLB entry *modified* bit, as well as fast *copy-on-write* page processing.

To speed processing of a copy-on-write or modified-bit usage, the faulting TLB entry is guaranteed flushed from the local CPU's TLB upon entry of this exception. Thus, in the common case, no flush operation needs to be generated before enabling write permission in the faulting TTE.

4.8 MMU fault status area

MMU related faults have their status and fault address information placed into a memory region made available by privileged code. Like the TSBs above, the fault status area for **each** virtual processor is declared via a hypervisor API call.

The MMU fault area is arranged on an aligned address boundary with instruction and data fault fields arranged into distinct 64byte blocks. The contents and layout of the MMU fault status area are currently specified in section12.6 of this specification.

5 Trap model

For sun4v, two of the three SPARC v9 trap types: precise and disrupting, behave according to the SPARC v9 specification. The third, deferred, may behave according to the UltraSPARC-I specification. The key difference is that UltraSPARC-I deferred traps do not provide additional information so that uncompleted instructions older than TPC can be emulated.

In the case of a CPU that implements SPARC v9 deferred traps, the hypervisor will present a deferred trap to privileged mode, but will also make available enough information so that privileged code can attempt to emulate any uncompleted instructions. In the case of a non-resumable error trap, the emulation information will appear in the error report. This is also the rationale for not including the SPARC v9 FQ register in sun4v, since it is used for emulation of deferred floating point traps.

A more precise description of the MMU, interrupt and error traps is made below to clarify behaviors left unspecified by SPARC v9.

5.1 Privilege mode trap processing

As with the SPARC v9 specification, the processor's action during trap processing depends on the trap type, the current trap level (TL register), and the processor state.

For trap processing from non-privileged or privileged mode to privileged mode the steps taken are the same as the SPARC v9 specification. Note that if a privileged code lowers the value of TL, there is no guarantee that the values of TSTATE, TPC, TNPC and TT will remain consistent for larger values of TL.

5.2 Trap levels

The maximum trap level available to privileged software in sun4v is defined to be 2 (MAXPTL).

5.2.1 Privilege mode TL overflow

When TL = MAXPTL, an additional privileged mode trap results in the delivery of a *watchdog_reset* trap to privileged mode with TT set to the type of trap that caused the error. TL remains at MAXPTL.

5.3 Sun4v privilege mode trap table

The privileged mode trap table is defined in the programmers reference manual for each specific processor.

6 Interrupt model

This chapter describes the sun4v architecture for sending and receiving interrupts.

6.1 Definitions

CPU mondo	CPU to CPU interrupt message.
Device mondo	interrupt sent by an I/O device.
Interrupt report	a message describing an interrupt
Interrupt queue	a FIFO list of interrupt reports

6.2 Interrupt reports

Interrupts are described by interrupt reports. Each interrupt report is 64 bytes long and consists of eight 64-bit words. If a report contains less than eight meaningful words it will be padded with zeros.

6.3 Interrupt queues

Interrupts are indicated to privileged mode via interrupt queues each with its own associated trap vector. There are 2 interrupt queues, one for device mondos and one other for CPU mondos. New interrupts are appended to the tail of a queue, and privileged code reads them from the head of the queue.

Privileged code is responsible for allocating real memory regions for these queues. Each queue region must be a power of 2 multiple of 64 bytes in size. The base real address must be aligned to the size of the region. For example, a queue of 128 entries is 8K bytes in size and must be aligned on an 8K byte real memory address boundary.

The queue configuration is described via hypervisor API calls when the queue region is created or modified (see section 11.2.6).

6.3.1 Queue support registers

The contents of each queue is described by a head and tail pointer. The head and tail pointer for each queue are held in registers as offsets from the base of their respective queue region. These interrupt queue registers are accessed with the QUEUE ASI (0x25). Each of the registers are addressable and accessible as 64bit quantities. The ASI addresses are as follows:

Register	Address	Access
CPU_MONDO_QUEUE_HEAD	0c3c0	rw
CPU_MONDO_QUEUE_TAIL	0x3c8	ro
DEV_MONDO_QUEUE_HEAD	0x3d0	rw
DEV_MONDO_QUEUE_TAIL	0x3d8	ro

In privileged mode, the head offset registers are read and write accessible, the tail offset registers are only readable. Attempting to write the tail register from privileged mode results in a *data_access_exception* trap.

*6.3.1.1 *_QUEUE_HEAD and *_QUEUE_TAIL*

The status of each queue is reflected by its head and tail pointers:

*_QUEUE_HEAD holds the offset to the oldest interrupt report in the queue.

*_QUEUE_TAIL holds the offset to the area where the next interrupt report will be stored.

An event that results in the insertion of a queue entry causes the tail of that queue to be incremented by 64 bytes. Privileged code is responsible for similarly incrementing the head pointer to remove an entry from the queue. The queue pointers are updated using modulo arithmetic based on the size of a queue. A queue is empty when the head is equal to the tail. A queue is full when the insertion of one more entry would cause the tail pointer to equal the head pointer.

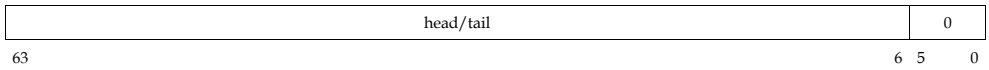


Figure 1 : Head and Tail register formats

The format of each of the QUEUE_HEAD and QUEUE_TAIL register is shown in Figure 1. Bits 0 through 5 always read as 0, and attempts to write them are ignored.

The minimum head and tail register size is provided as a property value in the machine description given to a guest.

6.4 Interrupt traps

The sun4v architecture has an interrupt trap for each of the two interrupt queues:

cpu_mondo this trap informs privileged mode that an interrupt report has been appended to the CPU mondo queue.

dev_mondo this trap informs privileged mode that an interrupt report has been appended to the dev mondo queue.

Both traps are disrupting, meaning that the current instruction stream can be restarted with a retry instruction, and that they can be blocked by setting pstate.ie = 0.

6.4.1 CPU mondo interrupts

CPU to CPU messages are are sent via CPU mondo interrupts. The term mondo refers to the original UltraSPARC-1 bus transaction where they were first introduced.

6.4.1.1 Sending CPU mondos

CPU mondos are sent via hypervisor API calls. The API allows 64 bytes of data to be sent to the targeted CPUs. The API call also includes the ability to send mondos to multiple CPUs in a single call to improve efficiency.

6.4.1.2 Receiving CPU mondos

CPU mondos are received via the CPU mondo queue.. When this queue is non-empty, a *cpu_mondo* disrupting trap is pended to the target CPU. The mondo data received is stored as the interrupt report.

6.4.2 Device mondo interrupts

Device mondo interrupts are received via the device mondo queue. When this queue is non-empty, a *dev_mondo* disrupting trap is pended to the target CPU. The interrupt report contents are device-specific, although a hypervisor API call does exist to allow privileged code to target device interrupts to specific CPUs.

7 Error model

This section describes the sun4v error handling and reporting architecture. To allow for a degree of future proofing, this component of sun4v has to be flexible, and robust enough to gracefully cope with error situations yet to be envisioned by system designers. In particular it is a design goal of sun4v that an older sun4v OS be able to handle reports from new hardware - if only via a set of default actions.

7.1 Definitions

Error class	a group of errors with common attributes that are handled in a similar manner.
Error report	a message describing an error sent to privileged mode.
Error queue	a FIFO list of error reports of the same class.

7.2 Error classes

The sun4v architecture defines two classes of errors: resumable and non-resumable errors.

7.2.1 Resumable error

A resumable error indicates the delivery of an error notification that leaves the current instruction stream in a consistent state so that execution can be resumed after the error is handled. A resumable error does not require any specific action by privileged code; the error may even be ignored. More sophisticated privileged code may record the error and/or forward it to a diagnosis agent. While all corrected errors are resumable, it is important to note that some uncorrectable errors are also resumable, e.g., an uncorrectable writeback error is resumable since the current instruction stream is not affected, but if the corrupted data is later fetched, a nonresumable error would occur. Whether or not the error was corrected is indicated in the error header.

7.2.2 Non-resumable error

A non-resumable error indicates the delivery of an error notification that leaves the current instruction stream in an inconsistent state. The instruction stream (nonprivileged or privileged) interrupted by this error cannot be resumed without explicit software intervention. In addition to possibly recording the error and/or forwarding it to a diagnosis agent, privileged code must either abort the current instruction stream, or attempt to recover from the error. The instruction stream may only be repaired if the error caused a precise trap. If the error caused a deferred trap, it cannot be repaired. The error's trap type is indicated in the error header.

7.3 Error reports

The sun4v architecture presents error information to privileged mode via error reports. An error report consists of a common 64 byte header, followed by error-specific data. The error-specific data will also be a multiple of 64 bytes in length, so the entire length of an error message will always be a multiple of 64 bytes.

7.4 Error queues

Errors are reported to privileged mode via error reports. Error reports are appended to a FIFO error queue. There are two error queues, one for each error class (resumable and non-resumable). Privileged code removes errors from the front of the error queue as it handles them.

The contents of each queue is described by a head and tail pointer. The head and tail pointer for each queue are held in registers as offsets from the base of their respective queue region. These interrupt queue registers are accessed with the QUEUE ASI (0x25). Each of the registers are addressable and accessible as 64bit quantities. The ASI addresses are as follows:

Register	Address	Access
RESUMABLE_ERROR_QUEUE_HEAD	0x3e0	read & write
RESUMABLE_ERROR_QUEUE_TAIL	0x3e8	read only
NONRESUMABLE_ERROR_QUEUE_HEAD	0x3f0	read and write
NONRESUMABLE_ERROR_QUEUE_TAIL	0x3f8	read only

In privileged mode, the head offset registers are read and write accessible, the tail offset registers are only readable. Attempting to write the tail register from privileged mode results in a *data_access_exception* trap.

7.4.1 *_QUEUE_HEAD and *_QUEUE_TAIL

The status of each queue is reflected by its head and tail pointers:

*_QUEUE_HEAD holds the offset to the oldest error report in the queue.

*_QUEUE_TAIL holds the offset to the area where the next error report will be stored.

An event that results in the insertion of a queue entry causes the tail of that queue to be incremented by 64 bytes. Privileged code is responsible for similarly incrementing the head pointer to remove an entry from the queue. The queue pointers are updated using modulo arithmetic based on the size of a queue. A queue is empty when the head is equal to the tail. A queue is full when the insertion of one more entry would cause the tail pointer to equal the head pointer.

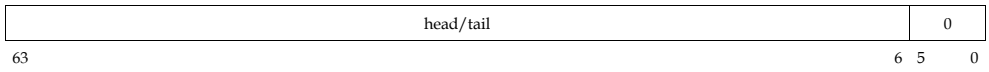


Figure 2 : Head and Tail register formats

The format of each of the QUEUE_HEAD and QUEUE_TAIL register is shown in Figure . Bits 0 through 5 always read as 0, and attempts to write them are ignored.

The minimum head and tail register size is 16 bits (bits 6 though 21). Unimplemented bits must read as zero, and be ignored when written.

7.5 Error traps

The sun4v architecture has two error traps:

resumable_error — this trap informs privileged code that an error report has been appended to the resumable error queue. This trap is a disrupting trap, meaning that the current instruction stream can be restarted with a retry instruction, and that *resumable_error* traps can be blocked by setting

pstate.ie = 0.

nonresumable_error this trap informs privileged code that an error report has been appended to the nonresumable error queue. This trap may be precise or deferred, as indicated in the error header. A precise trap may be restartable if the corruption can be repaired, but a deferred trap cannot be restarted even if the corruption is repaired. Non-resumable errors cannot be blocked, or nest. Privileged code must update the nonresumable queue head as quickly as possible to indicate when it is prepared to take another *nonresumable_error* trap. If the nonresumable_error queue is not empty when another *nonresumable_error* trap occurs, the hypervisor will stop the current CPU, and send a resumable error to another CPU in the same partition. If only one CPU has been configured in the partition, the hypervisor will inform the service processor.

At entry of the trap handler, the processor caches will be enabled and cleared of any faults. System memory, however, may have uncorrectable errors. If the real address of a memory error can be determined, this information will appear in the error header.

8 Machine description

To describe the resources within a virtual machine (or logical domain), a data structure called a machine description (MD) is made available to the guest running in each logical domain / virtual machine environment.

This section describes the transport format for the machine description (MD).

This format is provided for the contract between the producer of the MD (typically the Service Entity) and the consumers in the logical domains (for example, OBP boot firmware and the Solaris OS.)

8.1 Requirements

The format of the machine description is designed so that any consumer may either elect to read and transform it into an internal representation, or merely use it in place. For the latter, the encoding needs to be easily readable with an efficient decoder. Similarly a simple encoding requirement also exists for the system software responsible for generating a particular machine description.

A hypervisor will provide a machine description as a whole to a guest operating system upon request in response to an API call. The machine description is written into a buffer owned by the guest, and not shared with any other guest or with the hypervisor. Once provided it is truly private to the guest. Therefore, there is no requirement that the encoding format support any form of dynamic update or extension. Updates to a machine description are indicated by providing a complete new machine description.

8.2 Sections

The machine description is provided in four sections as illustrated below and described below.

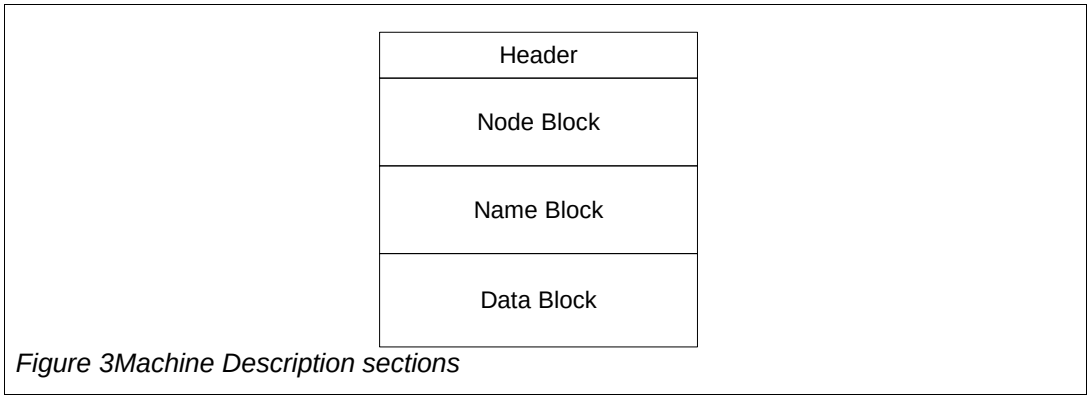


Figure 3Machine Description sections

These sections are linearly concatenated together to provide a single machine description.

8.3 Encoding

Unless otherwise specified, all fields described herein are encoded in network byte order (big-endian).

Unless otherwise specified, all fields are packed without intervening padding, and have no required byte alignment.

Where alignment is specified, it is defined in relation to the first byte of the machine description header.

8.4 Header

The format for the machine description header is defined below:

Byte offset	Size in bytes	Field name	Description
0	4	transport_version	Transport version number
4	4	node_blk_sz	Size in bytes of node block
8	4	name_blk_sz	Size in bytes of name block
12	4	data_blk_sz	Size in bytes of data block

The header is easily described by the following packed C structure for a big-endian machine:

```
struct MD_HEADER {
       uint32_t     transport_version;
       uint32_t     node_blk_sz;
       uint32_t     name_blk_sz;
       uint32_t     data_blk_sz;
};
```

The transport_version specifies the version encoding that applies to this MD. The transport version is a 32bit integer value. The upper 16bits correspond to a major version number, the lower 16bits correspond to a minor version number change.

8.4.1 Version numbering

The `transport_version` number for this specification is 0x10000, namely version 1.0.

An increase in the minor number of the transport version corresponds to the compatible addition or removal of information encoded in the machine description. This includes, but is not limited to, the removal of certain property types, or the addition of new property types. Guests can expect to be able to decode some, but not all of the Machine Description, and must handle this expectation accordingly by ignoring unknown types.

Future specification revisions defining new element types found outside a node encapsulation (e.g. between NODE_END and NODE) are considered incompatible and require an increase in the major version number of the MD transport header.

8.4.2 Size fields

- Each size field describes the size in bytes of the remaining three blocks in the machine description.
- The node block follows immediately after the section header.
- The name block starts at byte offset: 16+ node_blk_sz.
- The data block starts at byte offset: 16 + node_blk_sz + name_blk_sz.
- All sizes are multiples of 16 bytes.
- The total size of the MD is 16 + node_blk_sz + name_blk_sz + data_blk_sz.
- Each section (sizes; node_blk_sz, name_blk_sz, data_blk_sz) may be a maximum of 2^{32}-16 bytes in length.

Note: The name block and data block sections are described below first, to assist in understanding of the subsequent node block description.

8.5 Name Block

The name block provides name strings to be used for node entry naming. Legal name strings are defined as follows:

A name string is a human readable string comprised of an unaligned linear array of bytes (characters) terminated by a zero byte (nul '\0' character). Null termination enables the use of C functions such as `strcmp`(3) for comparison.

Character encoding consists of all human readable letters and symbols from ISO standard 8859-1 not including: blanks, "/", "\", ";", "[", "]", "@".

Each name string is referenced by its starting byte offset within the name block.

Name string lengths are stored along with the byte offset in the node elements, limiting name length to 255 bytes, not including the terminating null character.

There may not be duplicate strings in the name block; a given name string may appear only once in the name block. Thus the offset within the name block becomes a unique identifier for a given name string within a machine description.

A single name string may be referenced from more than one node element.

The name block is padded with zero bytes to ensure that the subsequent data block is aligned on a 16 byte boundary relative to the start of the machine description. These pad bytes are included in the name block size.

Note: The name block contains name strings that are held independently from the data block section in order to assist with accelerated string lookups. This technique is described later in section 24.2.

8.6 Data Block

The data block provides raw data that may be referenced by nodes in the node block.

Raw data associated with node block elements is simply a linear concatenation of the raw data itself and has no further intrinsic structure. The size, location and content of each data element is identified by the referring element in the node block.

Data block contents are unaligned unless specified as part of the referring property's requirements. When alignment is required it is considered relative to the first byte of the overall machine description. Alignment is achieved by preceeding a data element with zero bytes in the data block.

The producer of a machine description is required to arrange that data requiring a specific alignement in the MD is placed on an appropriate alignment boundary relative to the start of the MD. The consumer of an MD is required to read the machine description into a buffer aligned correctly for the largest alignment requirement the consumer may have, or be prepared to handle unaligned data references correctly.

8.7 Node Block

The node block is comprised of a linear array of 16 byte elements aligned on a 16byte boundary relative to the first byte of the entire machine description.

The node block elements have specific types and are grouped as defined below so as to form "nodes" of data. Each element is of fixed length, and each element may be uniquely identified by its index within the node block array.

Any element A may refer to another element B simply by using the array index for the location of element B. For example, the first element of the node block has index value 0, the second has index 1, and so on.

8.7.1 Element format

Elements within the node block have a fixed 16byte length format comprised of big-endian fields described below:

Byte offset	Size in bytes	Field name	Description
0	1	tag	Type of element
1	1	name_len	Length in bytes of element name. Element name is located in the name block.
2	2	_reserved_field	reserved field (contains bytes of value 0)
4	4	name_offset	Location offset of name associated with this element relative to start of name block.
8	8	val	64 bit value for elements of tag type "NODE", "PROP_VAL" or "PROP_ARC"
8	4	data_len	Length in bytes of data in data block for elements of type "PROP_STR" and of type "PROP_DATA"
12	4	data_offset	Location offset of data associated with this element relative to start of data block for elements of tag type "PROP_STR" and "PROP_DATA"

For a big-endian machine this is illustrated by the packed C structure below:

```
struct MD_ELEMENT {
       uint8_t      tag;
       uint8_t      name_len;
       uint16_t     _reserved_field;
       uint32_t     name_offset;
       union {
              struct  {
                     uint32_t     data_len;
                     uint32_t     data_offset;
              } y;
              uint64_t     val;
       } d;
};
```

The tag field defines how each element should be interpreted.

The name associated with this element is given by the name_offset and name_len fields giving the offset within the name block and length of the node name not including the terminating null character.

The remainder of the node element has two formats depending upon the node tag field. The node element either contains a 64bit immediate data value, or (for elements requiring an extended data or string) it consists of two 32bit values providing the size and offset of the relevant data within the data block.

8.7.2 Tag definitions

Note: Element tag enumerations are chosen so that an ASCII dump of the node section will reveal each element type thus aiding debugging.

The following element tag types are defined:

Tag Value	ASCII equiv	Name	Description	Value field
0x0	\0	LIST_END	End of element list	-
0x4e	'N'	NODE	Start of node definition	64bit index to next node in list of nodes
0x45	'E'	NODE_END	End of node definition	-
0x20	' '	NOOP	NOOP list element - to be ignored	0
0x61	'a'	PROP_ARC	Node property arc'ing to another node	64bit index of node referenced
0x76	'v'	PROP_VAL	Node property with an integer value	64bit integer value for property
0x73	's'	PROP_STR	Node property with a string value	offset and length of string within data block
0x64	'd'	PROP_DATA	Node property with a block of data	offset and length of property data with in the data block

8.8 Nodes

The array of elements in the node block form a sequence of "nodes" terminated by a single LIST_END element.

- A node is a linear sequence of two or more elements whose first element is NODE and whose last element is NODE_END.
- Between NODE and NODE_END there are zero or more elements that define properties for that node. These are PROP_* elements. The ordering of these elements (between NODE and NODE_END) does not confer meaning.
- The name given to a NODE element is non-unique and defines the binding of property elements that may be encapsulated within that node.
- The NOOP element is provided so that an entire node may be removed by overwriting all of its constituent elements with NOOP. A NODE link that arrives at a NOOP element is equivalent to the next NODE or LIST_END element after the sequence of NOOP elements.
- The PROP_ARC element is used to denote an arc in a DAG, therefore a PROP_ARC element may only reference a NODE element.
  - *Note: A node referenced by any PROP_ARC element cannot be removed by use of NOOP element unless all the referring PROP_ARC elements are removed. PROP_ARC elements may be removed by conversion to a NOOP element.*
- The element index of a "NODE" element is serves as a unique identification of a complete node and its encapsulated properties.
- The value field associated with a "NODE" element (elem_ptr->d.val) holds the element index to the next "NODE" element within the MD.

- *A reader may skip from one node to the next without having to scan within each node for the "NODE_END" by using this index value to locate the next NODE element in the node block.*

8.9 Node definitions

The type of a node is defined by the name string associated with the NODE element designating the start of the node in the machine description node block. Nodes can be found by linear search matching on type or by following the PROP_ARCs of a DAG.

8.9.1 Node categories

Nodes in a machine description serve one or two purposes; to provide information about a virtual machine resource they represent and, optionally to function as a construction node within a DAG formed within the machine description. A construction node may contain properties about certain resources, however its primary function is as a container for the arc links (PROP_ARC properties) that connect to other descriptive nodes.

Nodes belong to one of four categories that determine what walkers must handle within the MD. A node's category determines whether nodes of that type can be expected to found within the MD, or whether nodes of that type are optional. The categories are defined below:

core	Nodes of this type are always required to be present in the MD.
resource required	If the resource described by the node is available within the virtual machine, an associated node of this type is required to be present in the MD in order to describe the resource.
required by X	If a node of type X is present in the MD, then one (or more) nodes of this type will be present in the MD and associated with X.
optional	A node of this type need not appear as part of the MD, it is entirely optional, and guest OS code should have a default policy to continue functioning despite this absence.

8.10 Content versions

The "root" node (section 8.13.1) is unique in the entire machine description. It is; the one node from which all other nodes can be reached, guaranteed to be the first node defined in the node block, and is required to be present in a properly formed machine description.

The root node is primarily a construction node, with arc properties connecting to other nodes in the description. The root node carries a string property "content-version" that defines the version number of the content of the machine description".

Content versioning is defined independently of the machine description transport version. The content version identifies the rules surrounding construction of the DAG describing the machine.

This specification is for content version "1".

Minor changes such as the addition of new node types, properties or arc names, or the removal of optional nodes or properties, do not require a content version number change.

Incompatible changes to the node definitions such that any possible earlier machine description consumer will encounter problems with the newer content cause a version change.

8.11 Summary of node definitions

The list of currently defined nodes is as follows:

Node Name	Defined in section	Brief description
cache	8.14.1	Definition of a cache in the memory system hierarchy
cpu	8.13.3	Definition node for a single CPU
cpus	8.13.2	Construction node pointing to all cpu nodes
exec_unit	8.14.2	Node describing an execution unit of processor
mblock	8.13.5	Definition of single block of available memory
memory	8.13.4	Construction node pointing to all available mblock nodes
platform	8.13.6	Node describing intrinsic platform properties
root	8.13.1	The primary node
tlb	8.14.3	Definition of a TLB in the memory system heirarchy

Note: Nodes not defined in this specification must be ignored by system-software.

Each of the above nodes is defined in more detail in the following sections.

8.12 Common data definitions

As defined by the machine description transport, data values for string and data property elements (PROP_STR and PROP_DATA) are placed in the data block of the machine description. This section defines commonly used formats of data placed in the data block of a machine description and referred to using elements with the PROP_DATA tag.

Additional data formats may also be defined explicitly with a specific node definition.

8.12.1 String array

A string array is a commonly used data property that defines a concatenated list of nul character terminated strings. The PROP_DATA element that refers to this structure carries an offset (within the MD data block) to the start of the first string. The size field corresponds to a count of all the string bytes comprising the compound string list.

In this format strings are concatenated one immediately after the next. Thus if p is a pointer to the first string, then p+strlen(p)+1 is a pointer to the second. The overall size of this data field is used to determine the last string in the list. Every string in the list must terminate with the nul character. The string pointed to by p is the last string in the array if p+strlen(p)+1 equals the address of the property data plus its length. A string array of zero elements is not possible since the data length of a PROP_DATA element cannot be zero. Consumers should interpret the absence of the property as indicating an array of zero elements.

For example; the string list { "data","load","store" } would be encoded as a PROP_DATA pointing to a 16byte block of the data section of the MD with the byte values: 0x64 0x61 0x74 0x61 0x00 0x6c 0x6f 0x61 0x64 0x00 0x73 0x74 0x6f 0x72 0x65 0x00.

8.13 Generic nodes

8.13.1 Root node

Name	Category	Required subordinates	Optional subordinate
root	core	cpus (§8.13.2) memory (§8.13.4) platform (§8.13.6)	

8.13.1.1 Description

A node of this type must always be the first node in a machine description.

Only one node in the machine description may be named "root".

This root node must be the first node defined in the node block of the machine description.

All other nodes in the forward graph can be reached starting at the root node.

8.13.1.2 Properties

Name	Tag	Required	Description
content-version	PROP_STR	yes	Version string for the content of this machine description. Currently defined version is "1"

8.13.2 Cpus node

Name	Category	Required subordinates	Optional subordinate
cpus	required by root		cpu(§8.13.3)

8.13.2.1 Description

This construction node leads directly to all the virtual CPUs supported within this virtual machine. The number of cpus is expected to be derived by counting the number of subordinate cpu nodes.

8.13.2.2 Properties

None defined

8.13.3 Cpu node

Name	Category	Required subordinates	Optional subordinate
cpu	resource required		exec_unit (§8.14.2) cache (§8.14.1) tlb (§8.14.3)

8.13.3.1 Properties

Name	Tag	Required	Description
clock-frequency	PROP_VAL	yes	A 64-bit unsigned integer giving the frequency of the sun4v virtual CPU in Hertz and thereby the frequency of the processor's %tick register
compatible	PROP_DATA*	yes	String array of cpu types this virtual cpu is compatible with. The most specific cpu type must be placed first in the list, finishing with the least specific.
id	PROP_VAL	yes	A unique 64-bit unsigned integer identifier for the virtual CPU. This identifier is the one to use for all hypervisor CPU services for the CPU represented by this node.
isalist	PROP_DATA*	yes	List of the instruction set architectures supported by this virtual CPU.
mmu-#context-bits	PROP_VAL	no	A 64-bit unsigned integer giving the number of bits forming a valid context for use in a sun4v TTE and the MMU context registers for this virtual CPU. sun4v defines the minimum default value to be 13 if this property is not specified in a cpu node.
mmu-#shared-contexts	PROP_VAL	no	A 64-bit unsigned integer giving the number of primary and secondary shared context registers supported by this virtual CPU's MMU. If not present the default value is assumed to be 0
mmu-#va-bits	PROP_VAL	no	A 64-bit unsigned integer giving the number of virtual address bits supported by this virtual CPU. If not present a default value of 64 is assumed. Note: It is legal for there to be fewer VA bits than real address bits.
mmu-compatible	PROP_DATA*	no	String array listing alternate mmu-type values that this virtual CPU's MMU interface is also compatible with
mmu-max-#tsbs	PROP_VAL	no	A 64-bit unsigned integer giving the maximum number of TSBs this virtual CPU can simultaneously support. If not present the default value is assumed to be 1. *Note: sun4v Solaris assumes at least 2 are available.*
mmu-page-size-list	PROP_VAL	no	A 64-bit unsigned integer treated as a bit field describing the page sizes that may be used on this virtual CPU. Page size encodings are defined according to the sun4v TTE format (see §12.3.2). A bit N in this field, if set , indicates that sun4v defined page size with encoding N is available for use. For example bit 0 corresponds to the availability of 8K pages. If not present, a default value of 0x9 is assumed, indicating the sun4v default availability of 8K and 4M pages.
mmu-type	PROP_STR	yes	Name for the kind of MMU in use by this cpu Currently defined names are: “sun4v”
nwins	PROP_VAL	yes	A 64-bit unsigned integer giving the number of SPARCv9 register windows available on this virtual CPU

Name	Tag	Required	Description
q-cpu-mondo-#bits	PROP_VAL	yes	A 64-bit unsigned integer the maximum size (in bits) of the cpu mondo queue head and tail registers
q-dev-mondo-#bits	PROP_VAL	yes	A 64-bit unsigned integer giving the maximum size (in bits) of the device mondo queue head and tail registers
q-resumable-#bits	PROP_VAL	yes	A 64-bit unsigned integer giving the maximum size (in bits) of the resumable error queue head and tail registers
q-nonresumable-#bits	PROP_VAL	yes	A 64-bit unsigned integer giving the maximum size (in bits) of the non-resumable queue head and tail registers

Note: The 'compatible' will have "SUNW,sun4v" as the last element for systems of the sun4v machine class.

Note: Currently defined ISAs for constructing an 'islist' are: "sparcv9", "sparcv8plus", "sparcv8", "sparcv8-fsmuld", "sparcv7", "sparc".

8.13.4 Memory node

Name	Category	Required subordinates	Optional subordinate
memory	required by root		mblock(§8.13.5)

8.13.4.1 Description

This construction node leads directly to all the blocks of real address space backed by memory within this virtual machine.

8.13.4.2 Properties

None defined

8.13.5 Mblock node

Name	Category	Required subordinates	Optional subordinate
mblock	resource required		

8.13.5.1 Description

This node represents a single contiguous range of a virtual machine's real address space that is associated with real memory.

8.13.5.2 Properties

Name	Tag	Required	Description
base	PROP_VAL	yes	A 64-bit unsigned integer giving the base real address of the memory block represented bythis node
size	PROP_VAL	yes	A 64-bit unsigned integer giving the size in bytes of the memory block represented by this node

8.13.6 Platform node

Name	Category	Required subordinates	Optional subordinate
platform	core		

8.13.6.1 Description

This node holds general properties describing the platform a guest operating system is running on.

8.13.6.2 Properties

Name	Tag	Required	Description
banner-name	PROP_STR	yes	The banner name of the system.
hostid	PROP_VAL	no	A 64-bit unsigned integer in which the lower 32 bits hold the host id assigned to the virtual machine. The upper 32bits must be zero.
mac-address	PROP_VAL	no	A 64-bit unsigned integer in which the lower 48bits holds the mac address assigned to the virtual machine. The upper 16bits must be zero.
name	PROP_STR	yes	The platform binding name of the system. May not contain white space characters.
serial#	PROP_VAL	no	A 64-bit unsigned integer in which the lower 32 bits hold the serial number assigned to the virtual machine. The upper 32bits must be zero.
stick-frequency	PROP_VAL	yes	A 64-bit unsigned integer giving the frequency in Hertz of the system (%stick) clock for the virtual machine.
watchdog-resolution	PROP_VAL	no	The resolution, in milliseconds, of the watchdog API service. This property is present if the watchdog timer is service is available, but is otherwise not required.
watchdog-max-timeout	PROP_VAL	no	The largest number of milliseconds that is valid as a parameter to the watchdog timer service API. This property is present if the watchdog timer is service is available, but is otherwise not required.

Note: A platform's banner-name is cosmetic only, typically of the form "Sun Fire T100", but the name is part of the platform binding, typically of the form "SUNW,Sun-Fire-T100".

8.14 Memory hierarchy nodes

The following nodes are used to convey information about the host memory system heirarchy to a guest.

8.14.1 Cache node

Name	Category	Required subordinates	Optional subordinate
cache	optional		cache (§8.14.1)

8.14.1.1 Description

This node describes a cache in the memory system hierarchy.

8.14.1.2 Properties

Name	Tag	Required	Description
associativity	PROP_VAL	yes	A 64-bit unsigned integer giving the associativity of the cache (number of ways in each set). A value of 0 indicates fully associative, a value of 1 indicates direct-mapped, a value of 2 indicates 2-way and so on.
compatible-type	PROP_DATA	no	Holds a string array of "type" field values. In the event that a precise type match cannot be made using the "type" property this property may be searched for compatible types.
level	PROP_VAL	yes	A 64-bit unsigned integer giving the notional level of this cache in the memory hierarchy.
line-size	PROP_VAL	yes	A 64-bit unsigned integer giving the number of bytes comprising a single cache line. This is the size of the caches allocation unit that is matched by a single cache tag
sub-block-size	PROP_VAL	no	A 64-bit unsigned integer giving the number of bytes comprising a single cache sub-block. This is the size of the cache's coherence unit size that is matched by a single state entry. This property may be omitted if it would have the same value as the line-size property.
size	PROP_VAL	yes	A 64-bit unsigned integer giving the capacity (size) in bytes of the cache.
type	PROP_DATA	yes	String array listing what may be held in this cache. Generic types are "instruction" and "data".

8.14.2 Exec-unit node

Name	Category	Required subordinates	Optional subordinate
exec-unit	optional		cache (§8.14.1) tlb (§8.14.3)

8.14.2.1 Description

This node is describes an execution unit associated with a virtual CPU. Each execution unit may perform multiple functions/operations, and properties are defined appropriate not just to the whole execution unit, but also to individual function capabilities.

8.14.2.2 Properties

Name	Tag	Required	Description
compatible-type	PROP_DATA	no	If defined holds a string array of "type" field values. In the event that a precise type match cannot be made using the "type" property this property may be searched for compatible types.
type	PROP_DATA	yes	String array listing functional capabilities of this execution unit. Generic types are: "ifetch" - instruction fetcher "integer" - integer instruction execution "fp" - floating point instruction execution "vis" - vis instruction execution "integer-load" - integer load operations "integer-store" - integer store operations "fp-load" - floating point load operations "fp-store" - floating point store operations Niagara specific types are: "n1-crypto" - Niagara 1.0 crypto unit

8.14.3 TLB node

Name	Category	Required subordinates	Optional subordinate
tlb	optional		

8.14.3.1 Description

A TLB node describes a Translation Lookaside Buffer (MMU translation cache) in the memory system hierarchy.

8.14.3.2 Properties

Name	Tag	Required	Description
associativity	PROP_VAL	yes	A 64-bit unsigned integer giving the associativity of the TLB (number of ways in each set). A value of 0 indicates fully associative, a value of 1 indicates direct-mapped, a value of 2 indicates 2-way and so on.
compatible-type	PROP_DATA	no	If defined holds a string array of "type" field values. In the event that a precise type match cannot be made using the "type" property this property may be searched for compatible types.
entries	PROP_VAL	yes	A 64-bit unsigned integer giving the number of translation entries
level	PROP_VAL	yes	A 64-bit unsigned integer giving the notional level of this translation buffer in the overall page translation hierarchy
page-size-list	PROP_VAL	yes	A 64-bit unsigned integer treated as a bit field describing the page sizes that may be used in this TLB. Page size encodings are defined according to the sun4v Architecture Specification. A bit N in this field, if set , indicates that sun4v defined page size with encoding N is available for use. For example bit 0 corresponds to the availability of 8K pages.
type	PROP_DATA	yes	String array listing functional capabilities of this execution unit. Currently defined types are: "instruction" - translate instruction fetches "data" - translates data accesses

9 API versioning

This section describes the API versioning interface available to all privileged code.

9.1 API call

9.1.1 api_set_version

```
trap#              CORE_TRAP
function#          API_SET_VERSION
arg0               api_group
arg1               major_number
arg2               req_minor_number

ret0               status
ret1               act_minor_number
```

The API service enables a guest to request and check for a version of the Hypervisor APIs with which it may be compatible. It uses its own trap number to ensure consistency between future versions of the virtual machine environment. API services are grouped into sets that are specified by the argument *api_group*, (defined in the table below). For the specified group the guest's requested API major version number is given by the argument *major_number* and a requested API minor version number is given by the argument *req_minor_number*.

If the *major_number* is supported, the actual minor version implemented by the Hypervisor is returned in ret1 (*act_minor_number*). Note that the actual minor version number may be less than, equal to, or greater than the requested minor version number. (See Notes, below).

If the *major_number* is not supported, the Hypervisor returns an error code in ret0, and ret1 is undefined. (See Errors, below.)

If the *major_number* requested is zero, the version of the api_group selected is returned to the initial un-negotiated state, and the call will return with EOK in *status*, and *zero in act_minor_number*.

The version number of a specified API group may be set at any time with this API service, however;

1. The act of selecting an API version does not reset any previous state associated with services within a group.

2. Any API calls belonging to the same *api_group* being made concurrently with this api_set_version service will have undefined results.

3. Calls to APIs made concurrently with api_set_version that are not in api_group proceed as normally defined.

4. Simultaneous calls to api_set_version using the same api_group, may succeed but leave the api_group in an undefined state.

5. Simultaneous calls to api_set_version and api_get_version using the same api_group have undefined results for api_get_version.

The API groups are defined in Appendix B: Number Registry (on page 122) together with the approved version numbers for each of the API services defined in this specification.

Programming note: Each API group is treated independently of the others from a versioning perspective, so one group can have its version negotiated while APIs from other groups are actively being used. However, a guest operating system should take care to ensure that while a api_set_version is in progress, no APIs from the same api_group are used, and no other calls to api_set_version or api_get_version are made using the same api_group.

9.1.1.1 Errors

EINVAL	If api_group field is invalid or unsupported
ENOTSUPPORTED	If major number for that api_group is not supported
EOK	If api_group and major_number match, or major_number is zero

9.1.1.2 Usage Notes:

This API uses its own trap number, not for performance reasons, but to ensure its constancy even in the face of new API major versions.

Regardless of version number, the Hypervisor core APIs (CORE_TRAP) defined above enables any guest to print a message and cleanly exit its virtual machine environment in the event it is unsuccessful in negotiating an API version with which to communicate with other hypervisor functions.

The following informative text is provided as a guide to assist the reader in understanding the hypervisor versioning API.

API functions and returned data structures are categorized into specific groups. Each group represents an area of hypervisor functionality that may change independently of the others, and therefore may be versioned independently.

For each API group there is a major and a minor version number. Differences in the major version number indicate incompatible changes. Differences in the minor number indicate compatible changes, such that a higher version number espoused by the hypervisor will be compatible with a lower minor number requested by a guest. If the api_group is not supported the api_version function will return EINVAL. If the major version number for a valid api_group is not supported the api_version function will return ENOTSUPPORTED.

The handling of an unsupported API version is purely guest policy, however a guest may freely attempt a different major version if it is capable of driving that alternate interface. The suggested minimal behaviour is to print a warning message and exit the virtual machine.

By way of example consider a guest that requests minor version X, and this API may return minor version Y for a given *major_number* and *api_group*.

If X = Y, then the requested minor version is available.

If Y < X, the guest must be able to determine if the interface with minor version Y offers the required services and proceed accordingly. (This is a guest policy issue.)

If Y > X, then the guest may assume it can operate compatibly with version Y. Minor version number increments are defined to be compatible with the preceeding version, so in general the guest may accept Y when Y > X. In this case, the guest may want to print a warning, but that is up to the policy of the guest.

Alternatively in the event that Y>X, the hypervisor may elect to emulate version X, thus returning X.

9.1.2 api_get_version

```
trap#               CORE_TRAP
function#           API_GET_VERSION
arg0                api_group

ret0                status
ret1                major_number
ret2                minor_number
```

This service is used to determine the major and minor number of the most recently successfully negotiated API version for the specified group (see section 9.1.1). In the event that no API version has been successfully negotiated the call returns the error code EINVAL and ret1 and ret2 are set to 0.

9.1.2.1 Errors

```
EINVAL              - No API version yet successfully negotiated
```

10 Domain services

The following services enable privileged software to request information about or to affect the entire virtual machine domain.

10.1 API call

10.1.1 mach_exit

```
trap#               FAST_TRAP
function#           MACH_EXIT
arg0                exit_code
```

This service stops all CPUs in the virtual machine domain and places them into the *stopped* state. The 64-bit *exit_code* may be passed to a service entity as the domain's exit status.

On systems without a service entity, the domain will undergo a reset, and the boot firmware will be reloaded.

This function will never return to the guest that invokes it.

Note: by convention a exit_code of zero denotes successful exit by the guest code. A non-zero exit_code denotes a guest specific error indication.

10.1.1.1 Errors

This service does not return.

10.1.2 mach_desc

```
trap#               FAST_TRAP
function#           MACH_DESC
arg0                buffer
arg1                length

ret0                status
ret1                length
```

This service copies the most current machine description into the buffer indicated by the real address in arg0. The buffer provided must be 16 byte aligned. Upon success or EINVAL this service returns the actual size of the machine description is provided in the ret1 (length) return value.

Note: A method of determining the appropriate buffer size for the machine description is to first call this service with a buffer length of 0 bytes.

10.1.2.1 Errors

```
EBADALIGN           Buffer is badly aligned
ENORADDR            Buffer is to an illegal real address.
EINVAL              Buffer length is too small for complete machine
                    description.
```

10.1.3 mach_sir

```
trap#               FAST_TRAP
function#           MACH_SIR
```

This service provides a software initiated reset of a virtual machine domain. All CPUs are captured as soon as possible, all hardware devices are returned to the entry default state, and the domain is restarted at the SIR (trap type 0x4) real trap table (rtba) entry point on one of the CPUs. The single CPU restarted is selected as determined by platform specific policy. Memory is preserved across this operation.

10.1.3.1 Errors

This service does not return.

10.1.4 mach_set_soft_state

```
Trap#                FAST_TRAP
function#            MACH_SET_SOFT_STATE
arg0                 software_state
arg1                 software_description_ptr

ret0                 error code
```

This service enables the guest to report its soft state to the hypervisor. The soft state of the guest consists of two primary components: The first identifies whether the guest software is running or not. The second contains optional details specific to the software. The current soft state may be retrieved using the mach_get_soft_state API service.

The software_state argument is a 64-bit value used to indicate whether the guest software is operating normally or in a transitional state. The states "normal" and "in-transition" are defined in the Sun Indicator Standard.

```
SIS_NORMAL       0x1     guest software is operating normally
SIS_TRANSITION   0x2     guest software is in transition
```

The argument software_description_ptr is a real address of a data buffer of size 32 bytes aligned on a 32byte boundary. This buffer provides additional details specific to the guest software its operating state. The contents of this buffer are treated as a NUL terminated and padded 7-bit ASCII string of up to 31 characters not including the NUL termination.

The initial soft state is set to SIS_TRANSITION with an empty string for the software description.

10.1.4.1 Errors

```
EINVAL          - software_state is not valid, or
                    software_description is not NUL terminated
ENORADDR        - software_description is not a valid real addr buffer
EBADALIGNED     - software_description is not correctly aligned
```

10.1.4.2 Programming Notes

This service enables a guest operating system, or boot loader, to indicate its state to an entity external to the guest's virtual machine environment. Two simple states; "normal" or "transition" enable a guest to indicate whether it is operating normally, or in a transitional state such as booting or shutting down. The ability to provide a short message string enables the guest to supply additional human-readable information to supplement the two basic states.

Examples of this human readable string could be:

```
"OpenBoot before boot"
"OpenBoot booting"
"Solaris booting"
"Solaris panicked"
```

The virtual machine state is initially set to SIS_TRANSITION in the expectation that the guest operating environment will set the state to SIS_NORMAL once successfully started.

For example, while loading Solaris, OpenBoot may ignore, or set the state to transition several times (updating the informational string to identify different steps in the boot process), once booted and running Solaris may set the state to SIS_NORMAL indicating that it booted successfully. Similarly, when shutting down or panicking, Solaris may set the state to SIS_TRANSITION.

10.1.5 mach_get_soft_state

```
Trap#              FAST_TRAP
function#          MACH_GET_SOFT_STATE
arg0               software_description_ptr

ret0               error code
ret1               software_state
```

This service retrieves the current value of the guest's software state.

The software_description_ptr argument is the real address of a guest provided 32 byte buffer to be aligned on a 32 byte boundary. The API service will return the current value of the guest software description in this buffer. The hypervisor is only guaranteed to return up to and including the first NUL byte of the software description buffer contents (see mach_set_guest_state).

10.1.5.1 Errors

```
ENORADDR        - software_description is not a valid real addr buffer
EBADALIGNED     - software_description is not correctly aligned
```

10.1.6 mach_watchdog

```
trap#               FAST_TRAP
function#           MACH_WATCHDOG
arg0                timeout

ret0                status
ret1                time_remaining
```

This API service provides a basic watchdog timer service.

A guest uses this API to set a watchdog timer. Once the guest has set the timer, it must call the timer service again either to disable or postpone the expiration. If the timer expires before being reset or disabled, then the hypervisor takes a platform specific action leading to guest termination within a bounded time period. The platform action may include recovery actions such as reporting the expiration to a Service Processor, and/or automatically restarting the guest.

The *timeout* parameter is specified in milli-seconds, however the implemented granularity is given by the *watchdog-resolution* property in the *platform* node of the guest's machine description (see §8.13.6). The largest allowed *timeout* value is specified by the watchdog-max-timeout property of the *platform* node.

If the *timeout* argument is not zero, the watchdog timer is set to expire after a minimum of *timeout* milli-seconds.

If the *timeout* argument is zero, the watchdog timer is disabled.

If the *timeout* value exceeds the value of the *watchdog-max-timeout* property, the hypervisor leaves the watchdog timer state unchanged, and returns a status of EINVAL.

The *time_remaining* return value is valid regardless of whether the return status is EOK or EINVAL. A non-zero return value indicates the number of milli-seconds that were remaining until the timer was to expire. If less than one milli-second remains, the return value is 1. If the watchdog timer was disabled at the time of the call, the return value is 0.

Programming note: If the hypervisor cannot support the exact timeout value requested, but can support a larger timeout value, the hypervisor may round the actual timeout to a value larger than the requested timeout, consequently the time_remaining return value may be larger than the previously requested timeout value.

Programming note: Any guest OS debugger should be aware that the watchdog service may be in use. Consequently, it is recommended that the watchdog service is disabled upon debugger entry (e.g. reaching a breakpoint), and then re-enabled upon returning to normal execution. The API has been designed with this in mind, and the time_remaining result of the disable call may be used directly as the timeout argument of the re-enable call.

11 CPU services

CPUs represent devices that can execute software threads. A single chip that contains multiple cores or strands is represented as multiple CPUs with unique CPU identifiers. CPUs are exported to OBP via the machine description (and to Solaris via the device tree). CPUs are always in one of three states: *stopped, running,* or *error.*

11.1 CPU id and CPU list

A cpu id is a pre-assigned 16bit value that uniquely identifies a CPU within a logical domain.

Operations that are to be be performed on multiple CPUs specify them via a CPU list. A CPU list is an array in real memory, of which each 16-bit word is a CPU id.

CPU lists are passed through the API as two arguments: the first is the number of entries (16-bit words) in the CPU list, and the second is the (real address) pointer to the CPU id list.

11.2 API calls

11.2.1 cpu_start

```
trap#               FAST_TRAP
function#           CPU_START
arg0                cpuid
arg1                pc
arg2                rtba
arg3                target_arg0

ret0                status
```

Start CPU with id *cpuid* with *pc* in %pc and with a real trap base address value of r*tba*. The indicated CPU must be in the *stopped* state. The supplied rtba must be aligned on a 256byte boundary. On successful completion, the specified cpu will be in the *running* state and will be supplied with *target_arg0* in %o0 and *rtba* in %tba.

11.2.1.1 Errors

```
ENOCPU              Invalid cpuid
EINVAL              Target cpuid is not in the stopped state
ENORADDR            Invalid pc or rtba real address
EBADALIGN           Unaligned pc or unaligned rtba
EWOULDBLOCK         if starting resource is not available
```

11.2.2 cpu_stop

```
trap#               FAST_TRAP
function#           CPU_STOP
arg0                cpu

ret0                status
```

Stop CPU *cpu*. The indicated CPU must be in the *running* state. On completion, it will be in the *stopped* state. It is not legal to stop the current CPU.

Note: As this service cannot be used to stop the current cpu, this service may not be used to stop the last running CPU in a domain. To stop and exit a running domain a guest must use the mach_exit service.

11.2.2.1 Errors

```
ENOCPU              Invalid cpu
EINVAL              target cpu is the current cpu
EINVAL              target cpu is not in the running state
EWOULDBLOCK         if stopping resource is not available
ENOTSUPPORTED       if not supported on the platform
```

11.2.3 cpu_set_rtba

```
trap#               FAST_TRAP
function#           CPU_SET_RTBA
arg0                rtba

ret0                status
ret1                previous_rtba
```

Set the real trap base address of the local cpu to the value of r*tba*. The supplied rtba must be aligned on a 256byte boundary. Upon success the previous value of rtba is returned in ret1.

Note: the real trap table is described in the sun4v architecture specification.

Note: this service does not affect %tba

11.2.3.1 Errors

```
ENORADDR            Invalid rtba real address
EBADALIGN           rtba is incorrectly aligned for a trap table
```

11.2.4 cpu_get_rtba

```
trap#               FAST_TRAP
function#           CPU_GET_RTBA

ret0                status
ret1                previous_rtba
```

Returns the current value of rtba in ret1.

11.2.4.1 Errors

No possible error

11.2.5 cpu_yield

```
trap#              FAST_TRAP
function#          CPU_YIELD

ret0               status
```

Suspend execution on the current CPU. Execution may resume for any reason but is guaranteed to resume for any event that would generate a disrupting trap if pstate.ie=1.

11.2.5.1 Programming note:

This API may be used to save power and prevent contention on some CPUs by disabling hardware strands.

The guest is responsible for handling any race conditions that may occur when calling this service with pstate.ie=1.

Interrupts which are blocked by some mechanism other than pstate.ie (for example %pil) are not guaranteed to cause a return from this service.

11.2.5.2 Errors

No possible error

11.2.6 cpu_qconf

```
trap#              FAST_TRAP
function#          CPU_QCONF
arg0               queue
arg1               base raddr
arg2               nentries

ret0               status
```

Configure queue *queue* to be placed at real address *base,* and of *nentries* entries. nentries must be a power of two number of entries. B*ase* must be aligned exactly to match the queue size. Each queue entry is 64 bytes long, so for example, a 32 entry queue must be aligned on a 2048 byte real address boundary.

The specified queue is un-configured if nentries is 0.

For the current version of this API service the argument queue is defined as follows:

queue	**description**
0x3c	cpu mondo queue
0x3d	device mondo queue
0x3e	resumable error queue
0x3f	non-resumable error queue

Programming note: The maximum number of entries for each queue for a specific cpu may be determined from the machine description.

11.2.6.1 Errors

```
ENORADDR           Invalid base
EINVAL             Invalid queue or,
                   nentries not a power of two in number or,
                   nentries is less than two or too large.
EBADALIGN          baseaddr is not correctly aligned for size
```

11.2.7 cpu_qinfo

```
trap#               FAST_TRAP
function#           CPU_QINFO
arg0                queue

ret0                status
ret1                base raddr
ret2                nentries
```

Return the configuration info for queue *queue*. The base_raddr is the currently defined read address base of the defined queue, and nentries is the size of the queue in terms of number of entries.

For the current version of this API service the argument queue is defined as follows:

queue	**description**
0x3c	cpu mondo queue
0x3d	device mondo queue
0x3e	resumable error queue
0x3f	non-resumable error queue

If the specified queue is a valid queue number, but no queue has been defined this service will return success, but with nentries set to 0 and base_raddr will have an undefined value.

11.2.7.1 Errors

EINVAL	Invalid *queue*

11.2.8 cpu_mondo_send

```
trap#               FAST_TRAP
function#           CPU_MONDO_SEND
arg0-1              cpulist
arg2                data

ret0                status
```

Send a mondo interrupt to CPU list *cpulist* with 64 bytes of data pointed to by *data*. *data* must be a 64 byte aligned real address. The mondo data will be delivered to the cpu_mondo queues of the recipient cpus.

In all cases, (error or no), the cpus in *cpulist* to which the mondo has been successfully delivered will be indicated by having their entry in *cpulist* updated with the value 0xffff.

11.2.8.1 Errors

EBADALIGN	Mondo data is not 64byte aligned or cpulist is not 2byte aligned
ENORADDR	Invalid *data* mondo address, or invalid cpu list address
ENOCPU	Invalid CPU in cpus
EWOULDBLOCK	Some or all of the listed cpus did not receive the mondo
EINVAL	cpulist includes caller's cpuid

11.2.9 cpu_myid

```
trap#               FAST_TRAP
function#           CPU_MYID

ret0                status
ret1                cpuid
```

Return the hypervisor ID handle for the current CPU. Used by a virtual cpu to discover its own identity.

11.2.9.1 Errors

```
No errors defined
```

11.2.10 cpu_state

```
trap#               FAST_TRAP
function#           CPU_STATE
arg0                cpuid

ret0                status
ret1                state
```

Retrieve the current state of cpu *cpuid*. The states are:

```
CPU_STATE_STOPPED   0x1    cpu is in the stopped state
CPU_STATE_RUNNING   0x2    cpu is in the running state
CPU_STATE_ERROR     0x3    cpu is in the error state
```

11.2.10.1 Errors

```
ENOCPU              Invalid CPU in cpuid
```

12 MMU services

These hypervisor services control the behavior of address translations handled by the hypervisor.

A basic sun4v guest operating system, need not use any of these services at all. The default/initial operating environment for a guest is with virtual address translation disabled. In this mode all instructions and data references are made with real addresses.

If a guest operating system enables MMU translations, then virtual to real mappings may be specified in one of three different ways; either as permanent mappings, or as mappings that may be evicted and reloaded into system TLBs directly via MMU service functions, or indirectly via Translation Storage Buffers (TSBs). Moreover, with translations enabled, a guest Operating System must declare a Fault Status area for the hypervisor to provide information in the event of a translation fault.

12.1 Translation Storage Buffer (TSB) specification

The TSB functions control two sets of TSBs, one for when the virtual address context is zero, and one for when it is not zero. The demap functions remove translations from hardware TLBs.

A TSB description is a memory data structure that defines a single TSB:

offset	size	contents
0	2	page size to use for index shift in TSB
2	2	associativity of TSB
4	4	size of TSB in TTEs (16 bytes)
8	4	context_index
12	4	page size bitmask
16	8	real address of TSB base
24	8	reserved

The maximum TSB associativity supported is indicated in the guest machine description (see section 8.13.3).

12.1.1 Page sizes

The sun4v architecture defines value encodings of page size for translation table entries (TTEs). The page size bitmask indicates which of these encodings may be specified for TTEs within a given TSB. For each bit in the page size bitmask, if set, the sun4v page size may be specified. For example, bit 0 corresponds to an 8KByte page size, bit 1 to a 64K page size, and so on in multiples of 8 of the page size for each bit in the field:

Bit	Page size
0	8K
1	64K
2	512K
3	4MB
4	32MB
5	256MB
6	2GB
7	16GB

Bits 8 through 15 are reserved and must be set to zero.

The index shift page size indicates the page size to use for computing the TSB index for TTE retrieval. This value is the same as the page size value that may be specified in an individual sun4v TTE:

Value	Page size assumed for index computation
0	8K
1	64K
2	512K
3	4MB
4	32MB
5	256MB
6	2GB
7	16GB

Values 8 though 15 are reserved. The index shift value must correspond to the smallest page size specified in the page size bit mask.

12.1.2 Context index

This TSB description field enables TSBs to be defined where the context value for a page-translation is supplied within each entry of the TSB, or where a single value applies to the whole TSB. The latter enables a single TSB to be used for multiple context values (the context field within each TSB entry (TTE) is required to be zero). The context index field within a TSB description selects which of these two modes the TSB is defined to use.

If a context index field value of -1 (0xffffffff) is given in the TSB description, the TSB is defined to use the context field within each TTE.

If a context index field contains a value between 0 and *mmu-#shared-contexts*, the context value used for every entry in the TSB (TTE) will be taken from sun4v context register identified by the context index field at the time the TTE is used. For example, a translation required for (express or implied) ASI_PRIMARY and matched by a TTE in the TSB, will take its context value from the register PRIMARY_CONTEXT1 if the context index field of the TSB description is 1.

Any other value supplied in the context index field is invalid.

The value of *mmu-#shared-contexts* is provided in the *cpu* node (§8.13.3) of the machine description for each virtual cpu.

12.2 MMU flags

The MMU APIs are designed to function for both instruction and data address translations. Therefore, many of these interfaces take an MMU 'flags' argument in order to specify whether the operation is relevant to instruction or data mappings, or both. To ensure consistency between the MMU services this flags argument is defined here, and as follows:

The flags argument applies the API operation to instruction translations if bit 1 is set, and in addition applies the API operation to data translation entries if bit 0 is set. For every API service requiring a flags argument, at least one of bit 0 and/or bit 1 must be set.

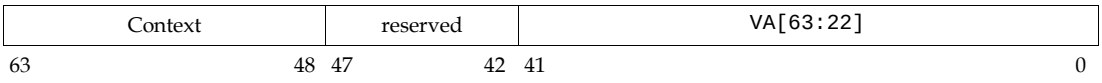
It is a programming error to request an instruction mapping (using the mapping flags) whose TTE's X bit is zero.

Implementation note: *For hardware implementations with unified instruction and data functions (for example; TLBs); Mapping an instruction translation entry may also cause an identical data translation entry to be mapped, and vice-versa even if not explicitly requests by the flags argument. Similarly, demapping an instruction translation entry may also cause the data translation entry to be demaped, and vice-versa even if not explicitly requested by the flags setting.*

12.3 Translation table entries

A TTE in a TSB describes virtual addresses to real address mappings.

TSB tag word



TSB data word

Figure 4 TSB entry (TTE) format

Sun4v specifies a TSB entry format with the following features:

12.3.1 TSB entry tag word

The 64bit TSB entry tag word has a 16bit context field, and a 42 bit VA field.

All 16-bits of the context field are significant. However, platforms are not required to support the full range (0 through 65535) of possible context values, thus certain context values are reserved and should not be used in the context field of the TSB entry tag. Use of a reserved context value results in a TSB entry miss. The guaranteed minimum range of supported context values is 0 through 8191. The availability of values between 8192 and 65535 is platform dependent. The maximum context value supported on a specific CPU is given in the machine description provided to a guest operating system.

The reserved field must be written as 0. Any non-zero values in this field will result in a TSB miss.

The VA field holds the upper 42bits of the virtual address to be matched for this TSB entry. All bits of this field are significant. For page sizes larger than 4MB, the appropriate lower VA address bits must be zero, or a TSB entry miss results.

Platforms are not required to support the full range of 64bit virtual addresses, however for platforms supporting fewer than 64 VA bits the highest order bit is sign-extended through bit 63 and compared with the entire VA field of the TTE entry tag word. This sign extension of virtual addresses results in a “hole” in the supported virtual address spaces. TSB entries whose VA tag fields fall within the hole will result in a TSB miss for that entry.

The range of virtual address bits supported for a specific CPU is given in the machine description provided to a guest operating system.

12.3.2 TSB entry data word

The sun4v TTE's range of the real address space is 56bits.

The UltraSPARC-1 TTE's lock bit has been removed from sun4v. Non faulting translation entries can be specified by privileged code via. a hypervisor API call.

The sun4v TTE data bitfields are as follows:

Bit Field	Mnemonic	Meaning
63	V	Valid. =1 if TTE is a valid entry
62	NFO	Non Faulting Only. If set to 1 this TTE is intended to match only loads using the non-faulting ASIs
61 - 56	SW	Software useable bits
55-13	RA	Real address bits 55 to 13. For page sizes larger than 8KB, the low order address bits below the page size are ignored
12	IE	Invert endianness
11	E	Side effect. If the side-effect bit it set, speculative loads will trap for addresses within the page, noncacheable memory addresses other than block loads and stores are strongly ordered against other E-bit accesses and non-cacheable stores are not merged. This bit should be set for pages that map I/O devices having side-effects. Note: the E bit does not prevent normal instruction prefetching. The E bit has no effect for instruction fetches. Note: The E bit does not force noncacheable access. It is expected, but not required that the CP and CV bits are cleared to 0 with the E bit. If both CP and CV are set ot 1 along with the E bit, the result is undefined Note: The E bit and the NFO bit are mutually exclusive: both bits should never be set in any TTE.
9 & 10	CP & CV	Cacheable Physical & Cacheable Virtual. These two bits are passed to the cache memory sub-system on any access and determine the cacheability of that access as follows: If CP is set to 1 then the mapped data or instructions may be cached in any physically indexed cache. If CP and CV are both set to 1 then the mapped data or instructions may be cached in any physically or virtually index cache. If CP is cleared to 0 then the contents of the mapped page are non-cacheable.
8	P	Privileged. If P is set to 1 then this mapping will only match in the TLB if the processor i s in privileged mode (PSTATE.priv = 1)
7	X	eXecute. If the X bits is set to 1 instructions may be fetched and executed from this page.

Bit Field	Mnemonic	Meaning
6	W	Writeable. If the W bit is set to 1, data mapped by this page may be written to.
5 & 4	SW	Software useable bits
3-0	Sz	Size: page size 0 = 8KB, 1=64KB, 2=512KB, 3=4MB, 4=32MB, 5=256MB, 6=2GB, 7=16GB Sizes 8 through 15 are reserved.

The size field of the sun4v TSB entry format is four bits wide. Page size values 0 through 7 are defined, while values 8 through 15 are reserved and should not be used. Attempts to specify page sizes in the range 8 through 15 result in an instruction_access_exception or data_access_exception indicating an invalid page size.

12.4 Translation storage buffer (TSB) configuration

TSBs are configured by privilege mode code via a hypervisor API call.

Each TSB can be configured in one of two different modes; context-match or context-ignore. The mode determines how a TSB entry is matched when the TSB is searched:

In context-match mode the context field of the TTE tag is matched against one of the nucleus, primary or secondary context registers (as specified by the actual or implied access ASI). This mode enables a TSB to be used for caching translation entries belonging to different contexts. Matching with the context field allows only those translations belonging to the current contexts to loaded into the TLB.

In context-ignore mode the context field of a TSB entry is ignored when the TSB is searched. A TSB configured in this mode must have the context field of each translation entry set to 0. When a valid TSB entry is matched it is loaded into the TLB with a context value provided from one of the primary or secondary context registers. The choice of primary or secondary is determined by the actual or implied access ASI, the index of the context register is specified as part of the TSB configuration. Context-ignore mode enables TSB entries to be used with more than one context.

Note: please refer to the section above on context registers, and in particular the possibility of multi-matching TLB entries.

12.5 Permanent and non-permanent mappings

It is an error to attempt to create overlapping permanent mappings. It is an error to create non-permanent mappings that conflict with permanent mappings. These errors are not necessarily detected, but may result in undefined behavior.

12.6 MMU Fault status area

MMU related faults have their status and fault address information placed into a memory region made available by privileged code. Like the TSBs above, the fault status area for **each** virtual processor is declared to the hypervisor via a hypervisor API call.

It is possible for MMU related faults to be delivered either by the hypervisor or directly by processor hardware if so implemented. For this reason, the MMU fault area is arranged on an aligned address boundary with instruction and data fault fields arranged into distinct 64byte blocks.

The layout of the MMU fault status area is described in the table below:

Offset (bytes)	Size (bytes)	Field
0x00	0x8	Instruction fault type (IFT)
0x08	0x8	Instruction fault address (IFA)
0x10	0x8	Instruction fault context A(IFC)
0x18	0x28	reserved
0x40	0x8	Data fault type (DFT)
0x48	0x8	Data fault address (DFA)
0x50	0x8	Data fault context (DFC)
0x58	0x28	reserved

The reserved fields must not be used. Their contents are undefined, and are not guaranteed preserved if written.

The definition of the values of the instruction and data fault type fields is as follows:

Code	Fault type
1	fast miss
2	fast protection
3	MMU miss
4	invalid RA
5	privileged violation
6	protection violation
7	NFO access
8	so page/NFO side effect
9	invalid VA
10	invalid ASI
11	nc atomic
12	privileged action
13	reserved
14	unaligned access
15	invalid page size
16 to -2	reserved
-1 (0xffffffffffffffff)	multiple errors

For each MMU related trap, the fault status area is updated as follows; (a blank entry for IFT,IFA,IFC,DFT,DFA or DFC indicates the field is not updated for the particular condition and is therefore undefined, and '●' indicates the field is updated with the relevant fault type, address or context information for the trap).

sun4v trap type	Fault type	IFT	IFA	IFC	DFT	DFA	DFC	Comments
instruction_access_exception	invalid RA (0x4)	●	●					instruction fetch to real address out of range
	privilege violation (0x5)	●	●	●				non privileged instruction access to privileged page (TTE.p=1)
	NFO access (0x7)	●	●	●				instruction access to non-faulting load page (TTE.nfo=1)
	invalid VA (0x9)	●	●	●				instruction virtual access out of range
	Invalid TSB entry	●	●	●				Hardware table walk found an invalid RA in a TTE loaded from a TSB
	Protection violation (0x6)	●	●	●				Instruction access to page without execute permission
	Multiple error (-1)	●						Hardware encountered multiple errors
instruction_access_MMU_miss	MMU miss (0x3)	●	●	●				TSB Miss
data_access_exception	invalid RA (0x4)				●	●	●	real address out of range
	privilege violation (0x5)				●	●	●	Non-privileged data access to privileged page (TTE.p=1)
	NFO access (0x7)				●	●	●	Data access to non-faulting page (TTE.nfo=1) with ASI other than a non-faulting ASI.
	so page/NFO side effect (0x8)				●	●	●	Non-faulting ASI data access to side-effect page (TTE.e=1)
	invalid VA (0x9)				●	●	●	Data or branch virtual access out of range
	invalid ASI (0xa)				●	●	●	Invalid ASI for instruction
	nc atomic (0xb)				●	●	●	Atomic access to non-cacheable page (TTE.cp=0)
	privileged action (0xc)				●	●	●	Data access by non-privileged software using a privileged or hyper-privileged ASI
	invalid page size (0xf)				●			
	Multiple error (-1)				●			Hardware encountered multiple errors
data_access_MMU_miss	MMU miss (0x3)				●	●	●	TSB Miss
data_access_protection	protection violation (0x6)				●	●	●	store to non-writeable ??

sun4v trap type	Fault type	IFT	IFA	IFC	DFT	DFA	DFC	Comments
mem_address_not_aligned LDDF_mem_address_not_aligned STDF_mem_address_not_aligned LDQF_mem_address_not_aligned STQF_mem_address_not_aligned	unaligned access (0xe)					●	●	Data access is not properly aligned
						●	●	
						●	●	
						●	●	
						●	●	
fast_instruction_access_MMU_miss	fast miss (0x1)		●	●				TLB Miss
fast_data_access_MMU_miss	fast miss (0x1)					●	●	TLB MIss
fast_data_access_protection	fast protection (0x2)					●	●	Store data access to page without write permission
privileged_action	privileged action (0xc)					●	●	Use of privileged ASI when pstate.priv = 0

12.7 API calls

12.7.1 mmu_tsb_ctx0

```
trap#               FAST_TRAP
function#           MMU_TSB_CTX0
arg0                ntsb
arg1                tsbdptr

ret0                status
```

Configures the TSBs for the current CPU for virtual addresses with context zero. *tsbdptr* is a pointer to an array of *ntsbs* TSB descriptions.

Note: the maximum number of TSBs available to a virtual CPU is given by the mmu-max-#tsbs property of the cpu's corresponding "cpu" node in the machine description.

12.7.1.1 Errors

```
ENORADDR            Invalid tsbdptr or TSB base in a TSB descriptor
EBADALIGN           tsbdptr is not aligned to an 8 byte boundary, or
                    TSB base in a descriptor is not aligned for a
                          TSB size
EBADPGSZ            Invalid pagesize in a TSB descriptor
EBADTSB             Invalid associativity or size in a TSB descriptor
EINVAL              Invalid ntsbs, or
                    invalid context index in a TSB descriptor, or
                    index page size not equal to smallest page size
                          in page size bitmask field.
```

12.7.2 mmu_tsb_ctxnon0

```
trap#               FAST_TRAP
function#           MMU_TSB_CTXNON0
arg0                ntsb
arg1                tsbdptr

ret0                status
```

Configures the TSBs for the current CPU for virtual addresses with non-zero contexts. *tsbdptr* is a pointer to an array of *ntsbs* TSB descriptions.

A maximum of 16 TSBs may be specified in the TSB description list.

12.7.2.1 Errors

```
ENORADDR            Invalid tsbdptr or TSB base in a TSB descriptor
EBADALIGN           tsbdptr is not aligned to an 8 byte boundary, or
                    TSB base in a descriptor is not aligned for a
                          TSB size
EBADPGSZ            Invalid pagesize in a TSB descriptor
EBADTSB             Invalid associativity or size in a TSB descriptor
EINVAL              Invalid ntsbs, or
                    invalid context index in a TSB descriptor, or
                    index page size not equal to smallest page size
                          in page size bitmask field.
```

12.7.3 mmu_demap_page

```
trap#               FAST_TRAP
function#           MMU_DEMAP_PAGE
arg0                reserved
arg1                reserved
arg2                vaddr
arg3                context
arg4                flags

ret0                status
```

Demaps any page mapping of virtual address *vaddr* in context *context* for the current virtual CPU. Any virtual tagged caches are guaranteed to be kept consistent. The flags argument is defined according to section 12.2; "MMU flags".

Arguments arg0 and arg1 are reserved and must be set zero.

12.7.3.1 Errors

The implementation of this function is not required to check for all possible errors, and may return the following error codes:

```
EINVAL              Invalid vaddr, context or flag value
ENOTSUPPORED        arg0 or arg1 is non-zero
```

12.7.4 mmu_demap_ctx

```
trap#               FAST_TRAP
function#           MMU_DEMAP_CTX
arg0                reserved
arg1                reserved
arg2                context
arg3                flags

ret0                status
```

Demaps all non-permanent virtual page mappings previously specified for context *context* for the current virtual CPU. Any virtual tagged caches are guaranteed to be kept consistent. The flags argument is defined according to section 12.2; "MMU flags".

Arguments arg0 and arg1 are reserved and must be set zero.

12.7.4.1 Errors

The implementation of this function is not required to check for all possible errors, and may return the following error codes:

```
EINVAL              Invalid context or flag value
ENOTSUPPORED        arg0 or arg1 is non-zero
```

12.7.5 mmu_demap_all

```
trap#              FAST_TRAP
function#          MMU_DEMAP_ALL
arg0               reserved
arg1               reserved
arg2               flags

ret0               status
```

Demaps all non-permanent virtual page mappings previously specified for the current virtual CPU. Any virtual tagged caches are guaranteed to be kept consistent. The flags argument is defined according to section 12.2; "MMU flags".

Arguments arg0 and arg1 are reserved and must be set zero.

12.7.5.1 Errors

The implementation of this function is not required to check for all possible errors, and may return the following error codes:

```
EINVAL             Invalid flag value
ENOTSUPPORED       arg0 or arg1 is non-zero
```

12.7.6 mmu_map_addr

```
trap#              MMU_MAP_ADDR
arg0               vaddr
arg1               context
arg2               TTE
arg3               flags

ret0               status
```

This API service creates a non-permanent mapping using the TTE to virtual address *vaddr* for *context* for the calling virtual CPU. The flags argument is defined according to section 12.2; "MMU flags".

Given a TTE specified with the valid bit clear, this service will have undefined behavior.

Note: This API call is for privileged code to specify temporary translation mappings without the need to create and manage a TSB.

12.7.6.1 Errors

The implementation of this function is not required to check for all possible errors, and may return the following error codes:

```
EINVAL             Invalid vaddr, context, or flag error
EBADPGSZ           Invalid page size value
ENORADDR           Invalid real address in TTE
```

12.7.7 mmu_map_perm_addr

```
trap#              FAST_TRAP
function#          MMU_MAP_PERM_ADDR
arg0               vaddr
arg1               reserved
arg2               TTE
arg3               flags

ret0               status
```

This API service creates a permanent mapping using the TTE to virtual address *vaddr* for the calling virtual CPU for context 0. The *reserved* field must be specified as zero.

A maximum of 8 such permanent mappings may be specified by privileged code. Mappings may be removed with **mmu_unmap_perm_addr** below.

This service guarantees an automatic demap of any conflicting non-permanent mappings.

It is an error to attempt to create overlapping permanent mappings. It is an error to create non-permanent mappings that conflict with existing permanent mappings.

The flags argument is defined according to section 12.2; “MMU flags“.

Given a TTE specified with the valid bit clear, this service will have undefined behavior.

Programming Notes:

This API call is used to specify address space mappings for which privileged code does not expect to receive misses. For example, this mechanism can be used to map kernel nucleus code and data.

To effect automatic de-map, this service may demap all non-permanent mappings.

12.7.7.1 Errors

```
EINVAL             Invalid vaddr, or flag error
EBADPGSZ           Invalid page size value
ENORADDR           Invalid real address in TTE
ETOOMANY           Too many mappings (maximum of 8 reached)
```

12.7.8 mmu_unmap_addr

```
trap#              MMU_UNMAP_ADDR
arg0               vaddr
arg1               context
arg2               flags

ret0               status
```

Demaps virtual address *vaddr* in context *context* on this CPU. This function is intended to be used to demap pages mapped with **mmu_map_addr**. This service is equivalent to invoking **mmu_demap_page** with only the current CPU in the CPU list.

The flags argument is defined according to section 12.2; “MMU flags“.

Attempting to perform an unmap operation for a previously defined permanent mapping will have undefined results.

12.7.8.1 Errors

The implementation of this function is not required to check for all possible errors, and may return the following error codes:

```
EINVAL             Invalid vaddr, context or flag value
```

12.7.9 mmu_unmap_perm_addr

```
trap#               FAST_TRAP
function#           MMU_UNMAP_PERM_ADDR
arg0                vaddr
arg1                reserved
arg2                flags

ret0                status
```

Demaps any permanent page mapping (established via mmu_map_perm_addr) of virtual address *vaddr* for context 0 for the current virtual CPU. Any virtual tagged caches are guaranteed to be kept consistent.

The flags argument is defined according to section 12.2; “MMU flags“.

12.7.9.1 Errors

```
EINVAL              Invalid vaddr or flag value
ENOMAP              Specified mapping was not found
```

12.7.10 mmu_fault_area_conf

```
trap#               FAST_TRAP
function#           MMU_FAULT_AREA_CONF
arg0                raddr

ret0                status
ret1                previous mmu fault area raddr
```

Configure the MMU fault status area for the calling CPU. A 64 byte aligned real address specifies where MMU fault status information is placed. The return value is the previously specified area, or 0 for the first invocation. Specifying a fault area at real address 0 is not allowed.

12.7.10.1 Errors

```
ENORADDR            Invalid real address
EBADALIGN           Invalid alignment for fault area
```

12.7.11 mmu_enable

```
trap#               FAST_TRAP
function#           MMU_ENABLE
arg0                enable_flag
arg1                return_target

ret0                status
```

This function either enables or disables virtual address translation for the calling CPU within the virtual machine domain. If the *enable_flag* is zero, translation is disabled, any non-zero value will enable translation.

When this function returns, the newly selected translation mode will be active. The argument *return_target* is a virtual address if translation is being enabled, or *return_target* is a real address in the event that translation is to be disabled.

Upon successful completion, this API service will return control to the *return_target* address with the new operating mode. In the event of call failure, the previous operating mode remains, and the service simply returns to the caller with the appropriate error code in ret0.

12.7.11.1 Errors

```
ENORADDR            Invalid real address when disabling translation
EBADALIGN           return_target is not aligned to an instruction
EINVAL              enable_flag requests current operating mode;
                    (e.g. disable if already disabled).
```

12.7.12 mmu_tsb_ctx0_info

```
trap#               FAST_TRAP
function#           MMU_TSB_CTX0_INFO
arg0                maxtsbs
arg1                bufferptr

ret0                status
ret1                ntsbs
```

This function returns the TSB configuration as previously defined by **mmu_tsb_ctx0** into the buffer provided by arg1. The size of the buffer is given in arg1 in terms of number of TSB description entries.

Upon return, ret1 always contains the number of TSB descriptions previously configured.

If zero TSBs were configured, then EOK is returned with ret1 containing 0.

12.7.12.1 Errors

```
EINVAL              supplied buffer (maxtsbs) is too small
EBADALIGN           bufferptr is badly aligned
ENORADDR            invalid real address for for buffer at bufferptr
```

12.7.13 mmu_tsb_ctxnon0_info

```
trap#               FAST_TRAP
function#           MMU_TSB_CTXNON0_INFO
arg0                maxtsbs
arg1                bufferptr

ret0                status
ret1                ntsbs
```

This function returns the TSB configuration as previously defined by **mmu_tsb_ctxnon0** into the buffer provided by arg1. The size of the buffer is given in arg1 in terms of number of TSB description entries.

Upon return ret1 always contains the number of TSB descriptions previously configured.

If zero TSBs were configured, then EOK is returned with ret1 containing 0.

12.7.13.1 Errors

```
EINVAL              supplied buffer (maxtsbs) is too small
EBADALIGN           bufferptr is badly aligned
ENORADDR            invalid real address for for buffer at bufferptr
```

12.7.14 mmu_fault_area_info

```
trap#               FAST_TRAP
function#           MMU_FAULT_AREA_INFO

ret0                status
ret1                fara
```

This API service returns the currently defined MMU fault status area for the current CPU. The real address of the fault status area is returned in ret1, or 0 is returned in ret1 if no fault status area is defined.

Note: mmu_fault_area_conf may be called with the return value (ret1) from this service if there is a need to save and restore the fault area for a cpu.

12.7.14.1 Errors

```
no errors are defined
```

13 Cache and Memory services

In general, caches and memory are not exposed to the supervisor, although they are described to it in the machine description.

13.1 API calls

13.1.1 mem_scrub

```
trap#               FAST_TRAP
function#           MEM_SCRUB
arg0                raddr
arg1                length

ret0                status
ret1                length scrubbed
```

This service zeros the memory contents for the memory address range raddr to raddr+length-1. It also creates a valid error-checking code for the memory address range raddr to raddr+length-1.

This service starts scrubbing at raddr, but may scrub less than length bytes of memory. On success the actual length scrubbed is returned in ret1.

The arguments raddr and length must be aligned to an 8K page boundary or must contain the start address and length from a sun4v error report.

Note: There are two uses for this function: The first use is to block clear and initialize memory and the second is to scrub an uncorrectable error reported via a resumable or non-resumable trap. The second use requires the arguments to be equal to the raddr and length provided in a sun4v memory error report.

13.1.1.1 Errors

ENORADDR	Invalid *raddr*
EBADALIGN	Either the start address or length are not correctly aligned.
EINVAL	*length* == 0

13.1.2 mem_sync

```
trap#               FAST_TRAP
function#           MEM_SYNC
arg0                raddr
arg1                length

ret0                status
ret1                length synced
```

For the memory address range *raddr* to *raddr*+*length*-1, this service forces the next access within that range to be fetched from main system memory.

This service starts syncing at *raddr*, but may sync less than *length* bytes of memory. On success the actual length synced is returned in ret1.

The arguments *raddr* and *length* must be aligned to an 8K page boundary.

13.1.2.1 Errors

```
ENORADDR            Invalid raddr
EBADALIGN           Either the start address or length are
                    not correctly aligned.
EINVAL              length == 0
```

14 Device interrupt services

Device interrupts are allocated to system bus bridges by the hypervisor, and described to the boot firmware in the machine description. OBP then describes them to Solaris via the device tree. The services described here are the generic interrupt services only, it is expected that the system bus nexus drivers will have additional APIs for functions that are specific to that bridge.

14.1 Definitions

These definitions apply to the following services:

cpuid — A unique opaque value which represents a target cpu.

devhandle — Device handle. The device handle uniquely identifies a sun4v device. It consists of the the lower 28-bits of the hi-cell of the first entry of the sun4v device's "reg" property as defined by the Sun4v Bus Binding to Open Firmware.

devino — Device interrupt number. Specifies the relative interrupt number within the device. The unique combination of devhandle and devino are used to identify a specific device interrupt.

Note: The devino value is the same as the values in the "interrupts" property or "interrupt-map" property in the sun4v device.

sysino — System Interrupt Number. A 64-bit unsigned integer representing a unique interrupt within a virtual machine.

intr_state — A flag representing the interrupt state for a given sysino. The state values are defined as:

Name	Value	Definition
INTR_IDLE	0	Nothing Pending
INTR_RECEIVED	1	Interrupt received by hardware
INTR_DELIVERED	2	Interrupt delivered to queue

intr_enabled — A flag representing the 'enabled' state for a given sysino. The state values are defined as:

Name	Value	Definition
INTR_DISABLED	0	sysino not enabled
INTR_ENABLED	1	sysino enabled

14.2 API calls

14.2.1 intr_devino_to_sysino

```
trap#              FAST_TRAP
function#          INTR_DEVINO2SYSINO
arg0               devhandle
arg1               devino

ret0               status
ret1               sysino
```

Converts a device specific interrupt number given by the arguments *devhandle* and *devino* into a system specific ino (*sysino*).

14.2.1.1 Errors

```
EINVAL             Invalid devhandle/devino
```

14.2.2 intr_getenabled

```
trap#              FAST_TRAP
function#          INTR_GETENABLED
arg0               sysino

ret0               status
ret1               intr_enabled
```

Returns state in *intr_enabled* for the interrupt defined by *sysino.* Return values are:

```
INTR_ENABLED or INTR_DISABLED
```

14.2.2.1 Errors

```
EINVAL             Invalid sysino
```

14.2.3 intr_setenabled

```
trap#              FAST_TRAP
function#          INTR_ENABLED
arg0               sysino
arg1               intr_enabled

ret0               status
```

Sets the 'enabled' state of the interrupt *sysino* legal values for *intr_enabled* are:

```
INTR_ENABLED or INTR_DISABLED
```

14.2.3.1 Errors

```
EINVAL             Invalid sysino or intr_enabled value
```

14.2.4 intr_getstate

```
trap#               FAST_TRAP
function#           INTR_GETSTATE
arg0                sysino

ret0                status
ret1                intr_state
```

Returns the current state of the interrupt given by the *sysino* argument.

14.2.4.1 Errors

```
EINVAL              Invalid sysino
```

14.2.5 intr_setstate

```
trap#               FAST_TRAP
function#           INTR_SETSTATE
arg0                sysino
arg1                intr_state

ret0                status
```

Sets the current state of the interrupt given by the *sysino* argument to the value given in the argument *intr_state*.

Note: Setting the state to INTR_IDLE clears any pending interrupt for *sysino*.

14.2.5.1 Errors

```
EINVAL              Invalid sysino or invalid intr_state
```

14.2.6 intr_gettarget

```
trap#               FAST_TRAP
function#           INTR_GETTARGET
arg0                sysino

ret0                status
ret1                cpuid
```

Returns the *cpuid* that is the current target of the interrupt given by the *sysino* argument.

The *cpuid* value returned is undefined if the target has not been set via *intr_settarget*.

14.2.6.1 Errors

```
EINVAL              Invalid sysino
```

14.2.7 intr_settarget

```
trap#               FAST_TRAP
function#           INTR_SETTARGET
arg0                sysino
arg1                cpuid

ret0                status
```

Set the target cpu for the interrupt defined by the argument *sysino* to the target cpu value defined by the argument *cpuid*.

14.2.7.1 Errors

```
EINVAL              Invalid sysino
ENOCPU              Invalid cpuid
```

15 Time of day services

The time of day (TOD) is maintained by the hypervisor on a per-domain basis. Setting the TOD in one domain does not affect any other domain.

Time is described by a single unsigned 64-bit word equivalent to a time_t for the POSIX time(2) system call. The word contains the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

15.1 API calls

15.1.1 tod_get

```
trap#               FAST_TRAP
function#           TOD_GET

ret0                status
ret1                time-of-day
```

Returns the current time-of-day. May block if TOD access is temporarily not possible.

15.1.1.1 Errors

```
EWOULDBLOCK         TOD resource is temporarily unavailable
ENOTSUPPORTED       If TOD not supported
```

15.1.2 tod_set

```
trap#               FAST_TRAP
function#           TOD_SET
arg0                tod

ret0                status
```

The current time-of-day is set to the value specified in arg0. May block if TOD access is temporarily not possible.

15.1.2.1 Errors

```
EWOULDBLOCK         TOD resource is temporarily unavailable
ENOTSUPPORTED       If TOD not supported
```

16 Console services

This section describes the API services provided for a guest console.

16.1 API calls

16.1.1 cons_getchar

```
trap#               FAST_TRAP
function#           CONS_GETCHAR

ret0                status
ret1                character
```

Returns a character from the console device. If no character is available then an EWOULDBLOCK error is returned. If a character is available, then the returned status is EOK and the character value is in ret1.

A virtual BREAK is represented by the 64-bit value -1.

A virtual HUP signal is represented by the 64-bit value -2.

16.1.1.1 Errors

```
EWOULDBLOCK         No character available
```

16.1.2 cons_putchar

```
trap#               FAST_TRAP
function#           CONS_PUTCHAR
arg0                char

ret0                status
```

This service sends a character to the console device. Only character values between 0 and 255 may be used. Values outside this range are invalid except as follows:

A virtual BREAK may be sent using the 64-bit value -1.

16.1.2.1 Errors

```
EINVAL              Illegal character
EWOULDBLOCK         Output buffer currently full, would block
```

17 Core dump services

When privileged code in a domain crashes/panics it may provide a capability to dump its internal state for later debugging. Such “core dumps” can be provided from the field to help diagnose field problems. However the hypervisor virtualizes much of the platform hardware, thus obscuring information about the physical resources that can be useful in diagnosing configuration related bugs.

Instead of adding a core dumping capability to the hypervisor, this API allows the domain's privileged code to dump platform and hypervisor-specific information as part of its own core dumping procedure. Privileged code allocates a section of its own memory space and informs the hypervisor that this may be used as a “dump buffer” for the hypervisor to place hypervisor specific debug/dump information.

Once declared, a dump buffer can be used at any time by the hypervisor to record private debug information, thus avoiding having such logs within the hypervisor itself.

The required size of the dump buffer is provided to the domain as part of the initial machine description.

During a core-dump operation, a guest requests that the hypervisor update any information in the dump buffer in preparation to being dumped as part of the domain's memory image.

Dump buffer information is highly platform and hypervisor specific. The format and content of the buffer are hypervisor private and should not be considered useable by sun4v code. Some platform hypervisors may provide no dump buffer information for security reasons.

17.1 API calls

17.1.1 dump_buf_update

```
trap#               FAST_TRAP
function#           DUMP_BUF_UPDATE
arg0                raddr
arg1                size

ret0                status
ret1                required size of dump buffer
```

This function declares a domain dump buffer to the hypervisor. The *raddr* supplies the real base address of the dump-buffer and must be 64-byte aligned.

The *size* field specifies the size of the dump buffer allocated, and may be larger than the minimum size specified in the machine description.

The hypervisor will fill the dump buffer with opaque data.

Note: a guest may elect to include dump buffer contents as part of a crash dump to assist with debugging. This function may be called any number of times so that a guest may relocate a dump buffer, or create "snapshots" of any dump-buffer information. Each call to `dump_buf_update` *atomically declares the new dump buffer to the hypervisor.*

A specified size of 0 unconfigures the dump buffer.

If *raddr* is an illegal or badly aligned real address, then any currently active dump buffer is disabled (equivalent to passing a size of 0) and an error is returned.

In the event that the call fails with EINVAL, ret1 contains the minimum size required by the hypervisor for a valid dump buffer.

17.1.1.1 Errors

```
ENORADDR            Invalid raddr
EBADALIGN           raddr not aligned on 64byte boundary
EINVAL              size is non-zero but less than minimum
                    size required
ENOTSUPPORTED       If not supported for current logical domain
```

17.1.2 dump_buf_info

```
trap#               FAST_TRAP
function#           DUMP_BUF_INFO

ret0                status
ret1                real address of current dump buffer
ret2                size of current dump buffer
```

This service returns the currently configured dump buffer description.

A returned size of 0 bytes indicates an undefined dump buffer. In this case the return address (ret1) is undefined.

17.1.2.1 Errors

```
No errors defined
```

18 Trap trace services

The hypervisor provides a trap tracing capability for privileged code running on each virtual CPU.

Privileged code provides a round-robin trap trace queue within which the hypervisor writes 64 byte entries detailing hyperprivileged traps taken on behalf of privileged code. This is provided as a debugging capability for privileged code.

18.1 Trap trace buffer control structure

The trap trace control structure is 64 bytes long and placed at the start (offset 0) of the trap trace buffer.

The format of the control structure is as follows:

```
Offset              Size   Field definition

0x00                8      Head offset
0x08                8      Tail offset
0x10                0x30   Reserved
```

The head offset is the offset of the most recently completed entry in the trap-trace buffer. The tail offset is the offset of the next entry to be written.

The control structure is owned and modified by the hypervisor. A guest may not modify the control structure contents. Attempts to do so will result in undefined behavior for the guest.

18.2 Trap trace buffer entry format

Trap trace entries all have the following format:

Offset	Size	Name	Description
0 x0	1	TTRACE_ENTRY_TYPE	Indicates hypervisor or guest entry
0x01	1	TTRACE_ENTRY_HPSTATE	Hyper-privileged state
0x02	1	TTRACE_ENTRY_TL	Trap level
0x03	1	TTRACE_ENTRY_GL	Global register level
0x04	2	TTRACE_ENTRY_TT	Trap type
0x06	2	TTRACE_ENTRY_TAG	Extended trap identifier
0x08	8	TTRACE_ENTRY_TSTATE	Trap state
0x10	8	TTRACE_ENTRY_TICK	Tick
0x18	8	TTRACE_ENTRY_TPC	Trap PC
0x20	8	TTRACE_ENTRY_F1	Entry specific
0x28	8	TTRACE_ENTRY_F2	Entry specific
0x30	8	TTRACE_ENTRY_F3	Entry specific
0x38	8	TTRACE_ENTRY_F4	Entry specific

For each entry the TTRACE_ENTRY_TYPE field value is defined as follows:

Value	Name	Description
0x00	TTRACE_TYPE_UNDEF	Entry content undefined
0x01	TTRACE_TYPE_HV	Hypervisor trap entry
0xff	TTRACE_TYPE_GUEST	Guest entry via ttrace_addentry service

18.3 API calls

18.3.1 ttrace_buf_conf

```
trap#              FAST_TRAP
function#          TTRACE_BUF_CONF
arg0               raddr
arg1               nentries

ret0               status
ret1               nentries
```

This function requests hypervisor trap tracing and declares a virtual cpu's trap trace buffer to the hypervisor. The *raddr* supplies the real base address of the trap trace queue and must be 64byte aligned.

The *nentries* field specifies the size in 64-byte entries of the buffer allocated. Specifying a value of zero for *nentries* disables trap tracing for the calling virtual cpu. The buffer allocated must be sized for a power of two number of 64 byte trap trace entries plus an initial 64 byte control structure.

This function may be called any number of times so that a virtual cpu may relocate a trap trace buffer, or create “snapshots” of information.

If *raddr* is an illegal or badly aligned real address, then trap tracing is disabled (equivalent to passing a *nentries* value of 0) and an error is returned.

Upon success ret1 is *nentries*.

Upon failure with EINVAL this service call returns in ret1 (*nentries*) the minimum number of buffer entries required.

Upon other failure ret1 is undefined.

18.3.1.1 Errors

```
ENORADDR           Invalid raddr
EINVAL             if size too small
EBADALIGN          raddr not aligned on 64byte boundary
```

18.3.2 ttrace_buf_info

```
trap#               FAST_TRAP
function#           TTRACE_BUF_INFO

ret0                status
ret1                raddr
ret2                size
```

This function returns the size and location of the previously declared trap-trace buffer. In the event that no buffer was previously declared, or the buffer disabled (e.g. via a ttrace_bufconf call with a size of zero), this call will return a size of zero (0) bytes.

18.3.2.1 Errors

none defined

18.3.3 ttrace_enable

```
trap#               FAST_TRAP
function#           TTRACE_ENABLE
arg0                enable

ret0                status
ret1                previous enable state
```

This function enables (or disables) trap tracing, returning the previously enabled state in ret1. Future systems may define various flags for the enable argument (arg0), for the moment a guest should pass (uint64_t)-1 to enable, and (uint64_t)0 to disable all tracing - which will ensure future compatibility.

18.3.3.1 Errors

```
EINVAL              No buffer currently defined
```

18.3.4 ttrace_freeze

```
trap#               FAST_TRAP
function#           TTRACE_FREEZE
arg0                freeze

ret0                status
ret1                previous_state
```

This function freezes (or unfreezes) trap tracing, returning the previous *freeze* state in ret1. A guest should pass a non-zero value to freeze and a zero value to un-freeze all tracing.

The returned previous_state is 0 for not frozen, and 1 for frozen.

18.3.4.1 Errors

```
EINVAL              No buffer currently defined
```

18.3.5 ttrace_addentry

```
trap#              TTRACE_ADDENTRY
arg0               tag (16-bits)
arg1               data word 0
arg2               data word 1
arg3               data word 2
arg4               data word 3

ret0               status
```

This function adds and entry to the trap trace buffer. Upon return only arg0/ret0 is modified - none of the other registers holding arguments are volatile across this hypervisor service.

18.3.5.1 Errors

```
EINVAL             No buffer currently defined
```

19 Logical Domain Channel services

The hypervisor provides communication channels to services and other domains. These channels are created by the Logical Domain Manager, and manifest themselves within a domain as an endpoint. Two endpoints are connected together and traffic is transferred by the hypervisor thus forming a logical domain channel (LDC).

19.1 Endpoints

Endpoints available within a domain are described within the Machine Description available via the MACH_DESC hypervisor API call. This API specification makes no assumptions about the peer on the other end of a LDC - the LDC APIs serve simply as a link communications layer with which higher level protocols are used for communication in and out of a logical domain. The details of these higher level protocols are usage specific and outside the scope of this link-layer specification.

Communication via an LDC occurs in the form of short fixed-length (64byte) message packets. Logical Domain Channels form bi-directional point-to-point links so all traffic sent to a local endpoint will arrive only at the corresponding endpoint at the other end of the channel. This fixed-length point-to-point nature means there is no address header or switching/routing operation performed by the hypervisor as part of packet delivery.

LDCs are not guaranteed as reliable link level communication channels. If a reliable or larger packet communication mechanism is required it must be provided as a protocol on top of this basic link-level communication mechanism.

19.2 LDC queues

LDC packets are delivered to an endpoint and deposited by the hypervisor into a queue provided by a guest operating system from its real address space. Only one receive queue may be allocated for each endpoint, and a channel direction is considered "down" while no receive queue is provided. Messages from a channel are deposited by the hypervisor at the "tail" of a queue, and the receiving guest indicates receipt by moving the corresponding "head" pointer for the queue.

A receive queue is defined to be consistent with other sun4v architecture queues, i.e. with the same restrictions as the cpu/device and error mondo queues. The guest identifies the queue to the hypervisor using an API call (LDC_RXQ_CONF) that is consistent with other queue API calls (for example CPU_QCONF). The head and tail pointers for an endpoint's receive queue are held by the hypervisor. Both the head and tail pointers are available via a hypervisor API call, but only the head pointer may be modified by a guest – also using a hypervisor API call.

To send LDC messages a guest operating system uses a transmit queue allocated from its own real address space. Only one transmit queue may be defined per-endpoint, undefined behavior for the sending guest occurs if the same memory is used for two or more different endpoint transmit queues. Like the receive queue, the transmit queue is defined to be consistent with other sun4v architecture queues such as the device and cpu mondo queues. The transmit queue's head and tail pointers are accessed via hypervisor API call.

To send a packet down an LDC, a guest deposits the packet into its transmit queue for the local endpoint, and then uses a hypervisor API call to update the tail pointer for the transmit queue. If an LDC is "up", then from the point at which a transmit queue becomes non-empty (a guest updates the tail pointer for its transmit queue), LDC packets are transferred from the transmit queue to the receive queue of the corresponding endpoint.

The assignment of a transmit queue does not affect whether an LDC is up or down.

19.3 LDC interrupts

To avoid the need for polling, LDC endpoints may be enabled to deliver interrupts to a guest domain indicating a change of endpoint state. Interrupts appear as mondos on the device mondo queue, with the mondo payload indicating the local LDC endpoint who's status has changed. The following endpoint states may be enabled to cause an interrupt;

LDC is down, LDC is up, receive queue is non-empty, receive queue is full, transmit queue is empty, transmit queue is not-full.

19.4 API calls

The following API calls are provided for LDC usage.

19.4.1 ldc_tx_qconf

```
trap#               FAST_TRAP
function#           LDC_TX_QCONF
arg0                ldc_id
arg1                base raddr
arg2                nentries

ret0                status
```

Configure transmit queue for LDC endpoint *ldc_id* to be placed at real address *base,* and of *nentries* entries. *nentries* must be a power of two number of entries. *Base_raddr* must be aligned exactly to match the queue size. Each queue entry is 64 bytes long, so for example, a 32 entry queue must be aligned on a 2048 byte real address boundary.

Upon configuration of a valid transmit queue the head and tail pointers are set to an hypervisor specific indentical value indicating that the queue initially is empty.

The endpoint's transmit queue is un-configured if *nentries* is 0.

Programming note: The maximum number of entries for each queue for a specific cpu may be determined from the machine description.

Programming note: A transmit queue may be specified even in the event that the LDC is down (peer endpoint has no receive queue specified). Transmission will begin as soon as the peer endpoint defines a receive queue.

Programming note: It is recommended that a guest wait for a transmit queue to empty prior to reconfiguring it, or un-configuring it. Re or un-configuration of a non-empty transmit queue behaves exactly as defined above, however it is undefined as to how many of the pending entries in the original queue will be delivered prior to the re-configuration taking effect. Furthermore, as the queue configuration causes a reset of the head and tail pointers there is no way for a guest to determine how many entries have been sent after the configuration operation.

19.4.1.1 Errors

ENORADDR	Invalid *base_raddr*
ECHANNEL	Invalid *ldc_id*
EINVAL	*nentries* not a power of two in number or, *nentries* is less than two or too large.
EBADALIGN	*base_raddr* is not correctly aligned for size

19.4.2 ldc_tx_qinfo

```
trap#               FAST_TRAP
function#           LDC_TX_QINFO
arg0                ldc_id

ret0                status
ret1                base raddr
ret2                nentries
```

Return the configuration info for the transmit queue of LDC endpoint *ldc_id.* The *base_raddr* is the currently defined real address base of the defined queue, and *nentries* is the size of the queue in terms of number of entries.

If the specified ldc_id is a valid endpoint number, but no transmit queue has been defined this service will return success, but with *nentries* set to 0 and *base_raddr* will have an undefined value.

19.4.2.1 Errors

```
ECHANNEL            Invalid ldc_id
```

19.4.3 ldc_tx_get_state

```
trap#               FAST_TRAP
function#           LDC_TX_GET_STATE
arg0                ldc_id

ret0                status
ret1                head_offset
ret2                tail_offset
ret3                channel_state
```

Return the transmit state, and the head and tail queue pointers for the transmit queue of LDC endpoint *ldc_id.* The head and tail values are the byte offset of the head and tail positions of the transmit queue for the specified endpoint.

The *channel_state* has the following defined values:

```
LDC_CHANNEL_DOWN    0
LDC_CHANNEL_UP      1
```

19.4.3.1 Errors

```
ECHANNEL            Invalid ldc_id
EINVAL              No transmit queue defined
EWOULDBLOCK         Operation would block
```

19.4.4 ldc_tx_set_qtail

```
trap#               FAST_TRAP
function#           LDC_TX_SET_QTAIL
arg0                ldc_id
arg1                tail_offset

ret0                status
```

Update the tail pointer for the transmit queue associated with the LDC endpoint ldc_id. The tail offset specified must be aligned on a 64byte boundary, and calculated so as to increase the number of pending entries on the transmit queue. Any attempt to decrease the number of pending transmit queue entries is considered an invalid tail offset and will result in an *EINVAL* error.

Programming note: Since the tail of the transmit queue may not be moved "backwards", the transmit queue may be "flushed" by configuring a new transmit queue, whereupon the hypervisor will configure the initial transmit head and tail pointers to be equal (queue empty).

19.4.4.1 Errors

ECHANNEL	Invalid *ldc_id*
EINVAL	No transmit queue defined, or invalid *tail_offset* value
EBADALIGN	*tail_offset* not correctly aligned
EWOULDBLOCK	Operation would block

19.4.5 ldc_rx_qconf

```
trap#               FAST_TRAP
function#           LDC_RX_QCONF
arg0                ldc_id
arg1                base raddr
arg2                nentries

ret0                status
```

Configure receive queue for LDC endpoint *ldc_id* to be placed at real address *base,* and of *nentries* entries. *nentries* must be a power of two number of entries. *Base_raddr* must be aligned exactly to match the queue size. Each queue entry is 64 bytes long, so for example, a 32 entry queue must be aligned on a 2048 byte real address boundary.

The endpoint's receive queue is un-configured if *nentries* is 0.

If a valid receive queue is specified for a local endpoint the LDC is in the up state for the purpose of transmission to this endpoint.

Programming note: The maximum number of entries for each queue for a specific cpu may be determined from the machine description.

Programming note: As receive queue configuration causes a reset of the queue's head and tail pointers there is no way for a guest to determine how many entries may have been received between a preceeding ldc_get_rx_state API call and the completion of the configuration operation. It should be noted that datagram delivery is not guaranteed via domain channels anyway, and therefore any higher protocol should be resilient to datagram loss if necessary. However, to overcome this specific race potential it is recommended, for example, that a higher level protocol be employed to ensure either re-transmission, or ensure that no datagrams are pending on the peer endpoint's transmit queue prior to the configuration operation.

19.4.5.1 Errors

ENORADDR	Invalid *base_raddr*
ECHANNEL	Invalid *ldc_id*
EINVAL	*nentries* not a power of two in number or, *nentries* is less than two or too large.
EBADALIGN	*base_raddr* is not correctly aligned for size

19.4.6 ldc_rx_qinfo

```
trap#               FAST_TRAP
function#           LDC_RX_QINFO
arg0                ldc_id

ret0                status
ret1                base raddr
ret2                nentries
```

Return the configuration info for the receive queue of LDC endpoint *ldc_id.* The *base_raddr* is the currently defined real address base of the defined queue, and *nentries* is the size of the queue in terms of number of entries.

If the specified ldc_id is a valid endpoint number, but no receive queue has been defined this service will return success, but with *nentries* set to 0 and *base_raddr* will have an undefined value.

19.4.6.1 Errors

```
ECHANNEL            Invalid ldc_id
```

19.4.7 ldc_rx_get_state

```
trap#               FAST_TRAP
function#           LDC_RX_GET_STATE
arg0                ldc_id

ret0                status
ret1                head_offset
ret2                tail_offset
ret3                channel_state
```

Return the receive state, and the head and tail queue pointers of the receive queue for LDC endpoint *ldc_id.* The head and tail values are the byte offset of the head and tail positions of the receive queue for the specified endpoint.

The *channel_state* has the following defined values:

```
LDC_CHANNEL_DOWN    0
LDC_CHANNEL_UP      1
```

19.4.7.1 Errors

```
ECHANNEL            Invalid ldc_id
EINVAL              No receive queue defined
EWOULDBLOCK         Operation would block
```

19.4.8 ldc_rx_set_qhead

```
trap#               FAST_TRAP
function#           LDC_RX_SET_QHEAD
arg0                ldc_id
arg1                head_offset

ret0                status
```

Update the head pointer for the receive queue associated with the LDC endpoint ldc_id. The head offset specified must be aligned on a 64byte boundary, and calculated so as to decrease the number of pending entries on the receive queue. Any attempt to increase the number of pending receive queue entries is considered an invalid head offset and will result in an *EINVAL* error.

Programming note: The receive queue may be"flushed" by setting the head offset equal to the current tail offset.

19.4.8.1 Errors

ECHANNEL	Invalid *ldc_id*
EINVAL	No receive queue defined, or invalid *head_offset* value
EBADALIGN	*head_offset* not correctly aligned
EWOULDBLOCK	Operation would block

20 PCI I/O Services

20.1 Introduction.

This section details Hypervisor services in support of PCI, PCI-X and PCI_Express interfaces.

20.1.1 External documents

The following documents are either referenced in this section, or should be consulted in together with this section

[1] sun4v Bus Binding to Open Firmware

[2] VPCI Bus Binding to Open Firmware

[3] PCI Express Base Specification 1.0a

20.2 IO Data Definitions

cpuid	A unique opaque value which represents a target cpu.
devhandle	Device handle. The device handle uniquely identifies a sun4v device. It consists of the the lower 28-bits of the hi-cell of the first entry of the sun4v device's "reg" property as defined by the Sun4v Bus Binding to Open Firmware.
devino	Device Interrupt Number. An unsigned integer representing an interrupt within a specific device.
sysino	System Interrupt Number. A 64-bit unsigned integer representing a unique interrupt within a "system".

20.3 PCI IO Data Definitions

devhandle	Device handle. The device handle uniquely identifies a sun4v device. It consists of the the lower 28-bits of the hi-cell of the first entry of the sun4v device's "reg" property as defined by the Sun4v Bus Binding to Open Firmware.
tsbnum	TSB Number. Identifies which io-tsb is used. For this version of the spec, tsbnum must be zero.
tsbindex	TSB Index. Identifies which entry in the tsb is is used. The first entry is zero.
tsbid	A 64-bit aligned data structure which contains a tsbnum and a tsbindex. bits 63:32 contain the tsbnum. bits 31:00 contain the tsbindex.
io_attributes	IO Attributes for iommu mappings. Attributes for iommu mappings. One or more of the following attribute bits stored in a 64-bit unsigned int.

```
PCI_MAP_ATTR_READ   0x01 - xfr direction is from memory
PCI_MAP_ATTR_WRITE  0x02 - xfr direction is to memory
```

Bits 63:2 are unused and must be set to zero for this version of the specification.

Note: For compatibility with future versions of this specification, the caller must set 63:2 to zero. The implementation shall ignore bits 63:2

r_addr — 64-bit Real Address.

pci_device — PCI device address. A PCI device address identifies a specific device on a specific PCI bus segment. A PCI device address is a 32-bit unsigned integer with the following format:

00000000.bbbbbbbb.dddddfff.00000000

Where:

bbbbbbbb is the 8-bit pci bus number

ddddd is the 5-bit pci device number

fff is the 3-bit pci function number

00000000 is the 8-bit literal zero.

pci_config_offset — PCI Configuration Space offset.

For conventional PCI, an unsigned integer in the range 0 .. 255 representing the offset of the field in pci config space.

For PCI implementations with extended configuration space, an unsigned integer in the range 0 .. 4095, representing the offset of the field in configuration space. Conventional PCI config space is offset 0 .. 255. Extended config space is offset 256 .. 4095

Note: For pci config space accesses, the offset must be 'size' aligned.

error_flag — Error flag

A return value specifies if the action succeeded or failed, where:

0 - No error occurred while performing the service.

non-zero - Error occurred while performing the service.

io_sync_direction — "direction" definition for pci_dma_sync

A value specifying the direction for a memory/io sync operation, The direction value is a flag, one or both directions may be specified by the caller.

```
0x01 - For device (device read from memory)
0x02 - For cpu (device write to memory)
```

io_page_list — A list of io_page_addresses. An io_page_address is an r_addr.

io_page_list_p — A pointer to an io_page_list.

"size based byte swap" - Some functions do size based byte swapping which allows sw to access pointers and counters in native form when the processor operates in a different endianness than the io bus. Size-based byte swapping converts a multi-byte field between big-endian format and little endian format as follows:

Size	Original value	Swapped value
2	0x0102	0x0201
4	0x01020304	0x04030201
8	0x0102030405060708	0x0807060504030201

20.4 API calls

The following APIs are provided for PCI services.

20.4.1 pci_iommu_map

```
trap#               FAST_TRAP
function#           PCI_IOMMU_MAP
arg0                devhandle
arg1                tsbid
arg2                #ttes
arg3                io_attributes
arg4                io_page_list_p

ret0                status
ret1                #ttes_mapped
```

Create iommu mappings in the sun4v device defined by the argument devhandle.

The mappings are created in the tsb defined by the tsbnum component of the tsbid argument. The first mapping is created in the tsb index defined by the tsbindex component of the tsbid argument. The call creates up to #ttes mappings, the first one at tsbnum, tsbindex, the second at tsbnum, tsbindex +1, etc.

All mappings are created with the attributes defined by the io_attributes argument.

The page mapping addresses are described in the io_page_list defined by the argument io_page_list_p, which is a pointer to the io_page_list. The first entry in the io_page_list is the address for the first iotte, the 2nd entry for the 2nd iotte, and so on.

Each io_page_address in the io_page_list must be appropriately aligned.

#ttes must be greater than zero.

For this version of the spec, the tsbnum component of the tsbid argument must be zero. Returns the actual number of mappings created, which may be less than or equal to the argument #ttes. If the function returns a value which is less than the #ttes, the caller may continue to call the function with an updated tsbid, #ttes, io_page_list_p arguments until all pages are mapped.

Note: This function does not imply an iotte cache flush. The guest must demap an entry before re-mapping it.

20.4.1.1 Errors

```
EINVAL              Invalid devhandle/tsbnum/tsbindex/io_attributes
EBADALIGN           Improperly aligned r_addr
ENORADDR            Invalid r_addr
```

20.4.2 pci_iommu_demap

```
trap#               FAST_TRAP
function#           PCI_IOMMU_DEMAP
arg0                devhandle
arg1                tsbid
arg2                #ttes

ret0                status
ret1                #ttes_demapped
```

Demap and flush iommu mappings in the device defined by the argument devhandle.

Demaps up to #ttes entries in the tsb defined by the tsbnum component of the tsbid argument, starting at the tsb index defined by the tsbindex component of the tsbid argument.

For this version of the spec, the tsbnum component of the tsbid argument must be zero.

#ttes must be greater than zero.

Returns the actual number of ttes demapped in the return value #ttes_demapped, which may be less than or equal to the argument #ttes. If #ttes_dempapped is less than #ttes, the caller may continue to call this function with updated tsbid and #ttes arguments until all pages are demapped.

Note: Entries do not have to be mapped to be demapped. A demap of an unmapped page will flush the entry from the tte cache.

20.4.2.1 Errors

```
EINVAL              invalid devhandle/tsbnum/tsbindex
```

20.4.3 pci_iommu_getmap

```
trap#               FAST_TRAP
function#           PCI_IOMMU_GETMAP
arg0                devhandle
arg1                tsbid

ret0                status
ret1                io_attributes
ret2                r_addr
```

Read and return the mapping in the device given by the argument devhandle and tsbid. If successful, the io_attributes shall be returned in ret1, the page address of the mapping shall be returned in ret2.

For this version of the spec, the tsbnum component of tsbid must be zero.

20.4.3.1 Errors

```
EINVAL              invalid devhandle/tsbnum/tsbindex
ENOMAP              Mapping is not valid - no translation exists
```

20.4.4 pci_iommu_getbypass

```
trap#               FAST_TRAP
function#           PCI_IOMMU_GETBYPASS
arg0                devhandle
arg1                r_addr
arg2                io_attributes

ret0                status
ret1                io_addr
```

Create a "special" mapping in the device given by the argument devhandle for the arguments given by r_addr and io_attributes. Return the io address in ret1 if successful.

Note: The error code ENOTSUPPORTED indicates that the function exists, but is not supported by the implementation.

20.4.4.1 Errors

```
EINVAL              Invalid devhandle/tsbnum/attributes
ENORADDR            Invalid real Address
ENOTSUPPORTED       Function not supported in this implementation.
```

20.4.5 pci_config_get

```
trap#               FAST_TRAP
function#           PCI_CONFIG_GET
arg0                devhandle
arg1                pci_device
arg2                pci_config_offset
arg3                size

ret0                status
ret1                error_flag
ret2                data
```

Read PCI configuration space for the pci adaptor defined by the argument devhandle.

Read size (1, 2 or 4) bytes of data for the PCI device defined by the argument pci_device, from the offset from the beginning of the configuration space defined by the argument pci_config_offset. If there was no error during the read access, set ret1 (error_flag) to zero and set ret2 to the data read. Insignificant bits in ret2 are not guaranteed to have any specific value and therefore must be ignored.

The data returned in ret2 is size based byte swapped.

If an error occurs during the read, set ret1 (error_flag) to a non-zero value.

pci_config_offset must be 'size' aligned.

20.4.5.1 Errors

```
EINVAL              invalid devhandle/pci_device/offset/size
EBADALIGN           pci_config_offset not size aligned
ENOACCESS           Access to this offset is not permitted
```

20.4.6 pci_config_put

```
trap#               FAST_TRAP
function#           PCI_CONFIG_PUT
arg0                devhandle
arg1                pci_device
arg2                pci_config_offset
arg3                size
arg4                data

ret0                status
ret1                error_flag
```

Write PCI config space for the pci adaptor defined by the argument devhandle.

Write 'size' bytes of data in a single operation. The argument 'size' must be 1, 2 or 4. The configuration space address is described by the arguments pci_device and pci_config_offset. pci_config_offset is the offset from the beginning of the configuration space given by the argument pci_device. The argument 'data' contains the data to be written to configuration space. Prior to writing the data is size based byte swapped.

If an error occurs during the write access, do not generate an error report, do set ret1 (error_flag) to a non-zero value. Otherwise, set ret1 to zero.

pci_config_offset must be 'size' aligned.

This function is permitted to read from offset zero in the configuration space described by the argument pci_device if necessary to ensure that the write access to config space completes.

20.4.6.1 Errors

```
EINVAL              invalid devhandle/pci_device/offset/size
EBADALIGN           pci_config_offset not size aligned
ENOACCESS           Access to this offset is not permitted
```

20.4.7 pci_peek

```
trap#               FAST_TRAP
function#           PCI_PEEK
arg0                devhandle
arg1                r_addr
arg2                size

ret0                status
ret1                error_flag
ret2                data
```

Attempt to read the io-address given by the arguments devhandle, r_addr and size. size must be 1, 2, 4 or 8. The read is performed as a single access operation using the given size. If an error occurs when reading from the given location, do not generate an error report, but return a non-zero value in ret1 (error_flag). If the read was successful, return zero in ret1 (error_flag) and return the actual data read in ret2 (data). The data returned in ret2 is size based byte swapped.

Non-significant bits in ret2 (data) are not guaranteed to have any specific value and therefore must be ignored. If ret1 (error_flag) is returned as non-zero, the data value is not guaranteed to have any specific value and should be ignored.

The caller must have permission to read from the given devhandle, r_addr, which must be an io address. The argument r_addr must be a size-aligned address.

The hypervisor implementation of this function must block access to any io address that the guest does not have explicit permission to access.

20.4.7.1 Errors

```
EINVAL              invalid size or devhandle
EBADALIGN           improperly aligned r_addr
ENORADDR            bad r_addr
ENOACCESS           guest access prohibited
```

20.4.8 pci_poke

```
trap#               FAST_TRAP
function#           PCI_POKE
arg0                devhandle
arg1                r_addr
arg2                size
arg3                data
arg4                pci_device

ret0                status
ret1                error_flag
```

Attempt to write data to the io-address described by the arguments devhandle, r_addr. The argument size defines the size of the 'write' in bytes and must be 1, 2 4 or 8.

The write is performed as a single operation using the given size. Prior to writing, the data is size based byte swapped.

If an error occurs when writing the data to the given location, do not generate an error report, but return a non-zero value in ret1 (error_flag). If the write operation was successful, return the value zero in ret1 (error_flag).

pci_device describes the configuration address of the device being written to. The implementation may safely read from offset 0 with the configuration space of the device described by devhandle and pci_device in order to guarantee that the write portion of the operation completes.

Any error that occurs due to the read shall be reported using the normal error reporting mechanisms .. the read error is not suppressed.

The caller must have permission to write to the given devhandle, r_addr, which must be an io address. The argument r_addr must be a size aligned address. The caller must have permission to read from the given devhandle, pci_device configuration space offset 0.

The hypervisor implementation of this function must block access to any io address that the guest does not have explicit permission to access.

20.4.8.1 Errors

```
EINVAL              invalid size, devhandle or pci_device
EBADALIGN           improperly aligned address
ENORADDR            bad address
ENOACESS            guest access prohibited
ENOTSUPPORTED       function is not supported by this implementation.
```

20.4.9 pci_dma_sync

```
trap#               FAST_TRAP
function#           PCI_DMA_SYNC
arg0                devhandle
arg1                r_adddr
arg2                size
arg3                io_sync_direction

ret0                status
ret1                #synced
```

Synchronize a memory region described by the arguments r_addr, size for the device defined by the argument devhandle using the direction(s) defined by the argument io_sync_direction. The argument size is the size of the memory region in bytes.

Return the actual number of bytes synchronized in the return value #synced, which may be less than or equal to the argument size. If the return value #synced is less than size, the caller must continue to call this function with updated r_addr and size arguments until the entire memory region is synchronized.

20.4.9.1 Errors

```
EINVAL              invalid devhandle or io_sync_direction
ENORADDR            bad r_addr
```

21 MSI Services

MSI services are effectively part of PCI, however, they are logically grouped into a separate set of services defined in this section.

21.1 Message Signaled Interrupt (MSI)

Message Signaled Interrupt as defined in the PCI Local Bus Specification and the PCI Express Base Specification. A device signals an interrupt via MSI using a posted write cycle to an address specified by system software using a data value specified by system software. The MSI capability data structure contains fields for the PCI address and data values the device uses when sending an MSI message on the bus. MSI-X is an extended form of MSI, but uses the same mechanism for signaling the interrupt as MSI. For the purposes of this document, the term "MSI" refers to MSI or MSI-X.

Root complexes that support MSI define an address range and set of data values that can be used to signal MSIs.

sun4v/pci requirements for MSI:

The root complex defines two address ranges. One in the 32-bit pci memory space and one in the 64-bit pci memory address space used as the target of a posted write to signal an MSI.

The root complex treats any write to these address ranges as signaling an MSI, however, only the data value used in the posted write signals the MSI.

21.2 MSI Event Queue (MSI EQ)

The MSI Event Queue is a page-aligned main memory data structure used to store MSI data records.

Each root port supports several MSI EQs, and each EQ has a system interrupt associated with it, and can be targeted (individually) to any cpu. The number of MSI EQs supported by a root complex is described by a property defined in [2]. Each MSI EQ must be large enough to contain all possible MSI data records generated by any one PCI root port. The number of entries in each MSI EQ is described by a property defined in [2].

Each MSI EQ is compliant with the definition of interrupt queues described in [4], however, instead of accessing the queue head/tail registers via ASI-based registers, an API is provided to access the head/tail registers.

The sun4v/pci compliant root complex has the ability to generate a system interrupt when the MSI EQ is non-empty.

MSI/Message/INTx Data Record format

Each data record consists of 64 bytes of data, aligned on a 64-byte boundary.

The data record is defined as follows:

```
666655555555554444444444333333333322222222221111111110000000000
321098765432109876543210987654321098765432109876543210987654321098765432109876543210

0x00:
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVxxxxxxxxxxxxxxxxxxxxxxxxTTTTTTTT

0x08:
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
```

```
0x10:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

0x18:
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS

0x20:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxRRRRRRRRRRRRRRR

0x28:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

0x30:
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD

0x38:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Where,

xx..xx are unused bits and must be ignored by sw.

VV..VV is the version number of this data record

For this release of the spec, the version number

field must be zero.

TTTTTTTT is the data record type:

Upper 4 bits are reserved, and must be zero

```
0000 - Not an MSI data record - reserved for sw use.
0001 - MSG
0010 - MSI32
0011 - MSI64
0010 - Reserved
...
0111 - Reserved
1000 - INTx
1001 - Reserved
                          ...
1110 - Reserved
1111 - Not an MSI data record - reserved for sw use.
```

All other encodings are reserved.

II..II is the sysino for INTx (sw defined value), otherwise zero.

SS..SS is the message timestamp if available.

If supported by the implementation, a non-zero value in this field is a copy of the %stick register at the time the message is created. If unsupported, this field will contain zero.

RR..RR is the requester ID of the device that initiated the MSI/MSG and has the following format:

bbbbbbbb.dddddfff

Where bb..bb is the bus number, dd..dd is the device number and fff is the function number.

Note that for PCI devices or any message where the requester is unknown, this may be zero, or the device-id of an intermediate bridge.

For intx messages, this field should be ignored.

AA..AA is the MSI address. For MSI32, the upper 32-bits must be zero. (for data record type MSG or INTx, this field is ignored)

DD..DD is the MSI/MSG data or INTx number

For MSI-X, bits 31..0 contain the data from the MSI packet which is the msi-number. bits 63..32 shall be zero.

For MSI, bits 15..0 contain the data from the MSI message which is the msi-number. bits 63..16 shall be zero

For MSG data, the message code and message routing code are encoded as follows:

```
63:32 - 0000.0000.0000.0000.0000.0000.GGGG.GGGG
32:00 - 0000.0000.0000.0CCC.0000.0000.MMMM.MMMM
```

Where,

GG..GG is the target-id of the message in the

following form:

bbbbbbbb.dddddfff

where bb..bb is the target bus number.

ddddd is the target deviceid

fff is the target function number.

CCC is the message routing code as defined by [3]

MM..MM is the message code as defined by [3]

For INTx data, bits 63:2 must be zero and the low order 2 bits are defined as follows:

```
00 - INTA
01 - INTB
10 - INTC
11 - INTD
```

21.3 Definitions

cpuid	A unique opaque value which represents a target cpu.
devhandle	Device handle. The device handle uniquely identifies a sun4v device. It consists of the the lower 28-bits of the hi-cell of the first entry of the sun4v device's "reg" property as defined by the Sun4v Bus Binding to Open Firmware.
msinum	A value defining which MSI is being used.
msiqhead	The offset value of a given MSI-EQ head.
msiqtail	The offset value of a given MSI-EQ tail.

msitype Type specifier for MSI32 or MSI64

```
0 - type is MSI32
1 - type is MSI64
```

msiqid A number from 0 .. 'number of MSI-EQs - 1', defining which MSI EQ within the device is being used.

msiqstate An unsigned integer containing one of the following values:

```
PCI_MSIQSTATE_IDLE        0           # idle (non-error) state
PCI_MSIQSTATE_ERROR       1           # error state
```

msiqvalid An unsigned integer containing one of the following values:

```
PCI_MSIQ_INVALID          0           # disabled/invalid
PCI_MSIQ_VALID            1           # enabled/valid
```

msistate An unsigned integer containing one of the following values:

```
PCI_MSISTATE_IDLE         0           # idle/not enabled
PCI_MSISTATE_DELIVERED    1           # MSI Delivered
```

msivalid An unsigned integer containing one of the following values:

```
PCI_MSI_INVALID           0           # disabled/invalid
PCI_MSI_VALID             1           # enabled/valid
```

msgtype A value defining which MSG type is being used. An unsigned integer containing one of the following values: (as per PCIe spec 1.0a)

```
PCIE_PME_MSG              0x18        PME message
PCIE_PME_ACK_MSG          0x1b        PME ACK message
PCIE_CORR_MSG             0x30        Correctable message
PCIE_NONFATAL_MSG         0x31        Non fatal message
PCIE_FATAL_MSG            0x33        Fatal message
```

msgvalid An unsigned integer containing one of the following values:

```
PCIE_MSG_INVALID          0           # disabled/invalid
PCIE_MSG_VALID            1           # enabled/valid
```

21.4 API calls

21.4.1 pci_msiq_conf

```
trap#               FAST_TRAP
function#           PCI_MSIQ_CONF
arg0                devhandle
arg1                msiqid
arg2                r_addr
arg3                nentries

ret0                status
```

Configure the MSI queue given by the arguments *devhandle, msiqid* for use and to be placed at real address *r_addr,* and of *nentries* entries. *nentries* must be a power of two number of entries.

r_addr must be aligned exactly to match the queue size. Each queue entry is 64 bytes long, so for example, a 32 entry queue must be aligned on a 2048 byte real address boundary.

The MSI-EQ Head and Tail are initialized so that the MSI-EQ is 'empty'.

Implementation Note: Certain implementations have fixed sized queues. In that case *nentries* must contain the correct value.

21.4.1.1 Errors

```
EINVAL              Invalid devhandle, msiqid or nentries
EBADALIGN           improperly aligned r_addr
ENORADDR            bad r_addr
```

21.4.2 pci_msiq_info

```
trap#               FAST_TRAP
function#           PCI_MSIQ_CONF
arg0                devhandle
arg1                msiqid

ret0                status
ret1                r_addr
ret2                nentries
```

Return configuration information for the MSI queue given by the arguments *devhandle, msiqid*.

The base address of the queue is returned in *r_addr.* The number of entries in the queue is returned in *nentries*.

If the queue is unconfigured *r_addr* is undefined and returns zero in *nentries*.

21.4.2.1 Errors

```
EINVAL              Invalid devhandle or msiqid
```

21.4.3 pci_msiq_getvalid

```
trap#              FAST_TRAP
function#          PCI_MSIQ_GETVALID
arg0               devhandle
arg1               msiqid

ret0               status
ret1               msiqvalid
```

Get the valid state of the MSI-EQ defined by the arguments *devhandle* and *msiqid*.

21.4.3.1 Errors

```
EINVAL                    bad devhandle or msiqid
```

21.4.4 pci_msiq_setvalid

```
trap#              FAST_TRAP
function#          PCI_MSIQ_SETVALID
arg0               devhandle
arg1               msiqid
arg2               msiqvalid

ret0               status
```

Set the valid state of the MSI-EQ defined by the arguments *devhandle* and *msiqid* to the state described by the argument *msiqvalid*. *msiqvalid* must be PCI_MSIQ_VALID or PCI_MSIQ_INVALID.

21.4.4.1 Errors

EINVAL bad *devhandle* or *msiqid* or *msiqvalid* value or MSI EQ is uninitialized.

21.4.5 pci_msiq_getstate

```
trap#              FAST_TRAP
function#          PCI_MSIQ_GETSTATE
arg0               devhandle
arg1               msiqid

ret0               status
ret1               msiqstate
```

Get the state of the MSI-EQ defined by the arguments *devhandle* and *msiqid*.

21.4.5.1 Errors

```
EINVAL                    bad devhandle or msiqid
```

21.4.6 pci_msiq_setstate

```
trap#               FAST_TRAP
function#           PCI_MSIQ_SETSTATE
arg0                devhandle
arg1                msiqid
arg2                msiqstate

ret0                status
```

Set the state of the MSI-EQ defined by the arguments *devhandle* and *msiqid* to the state described by the argument *msiqstate*. *msiqstate* must be PCI_MSIQSTATE_IDLE or PCI_MSIQSTATE_ERROR.

21.4.6.1 Errors

```
EINVAL              bad devhandle, msiqid or msiqstate or
                    MSI EQ is uninitialized.
```

21.4.7 pci_msiq_gethead

```
trap#               FAST_TRAP
function#           PCI_MSIQ_GETHEAD
arg0                devhandle
arg1                msiqid

ret0                status
ret1                msiqhead
```

Return the current *msiqhead* for the MSI-EQ described by the argument *devhandle, msiqid*.

21.4.7.1 Errors

```
EINVAL              Invalid devhandle or msiqid or
                    MSI EQ uninitialized
```

21.4.8 pci_msiq_sethead

```
trap#               FAST_TRAP
function#           PCI_MSIQ_GETHEAD
arg0                devhandle
arg1                msiqid
arg2                msiqhead

ret0                status
```

Set the MSI EQ queue head in the MSI EQ described by the arguments *devhandle, msiqid* to the value given by the *msiqhead* argument.

21.4.8.1 Errors

```
EINVAL              Invalid devhandle, msiqid or msiqhead
                    or MSI EQ is uninitialized
```

21.4.9 pci_msiq_gettail

```
trap#              FAST_TRAP
function#          PCI_MSIQ_GETTAIL
arg0               devhandle
arg1               msiqid

ret0               status
ret1               msiqtail
```

Return the current *msiqtail* for the MSI-EQ described by the argument *devhandle, msiqid*.

21.4.9.1 Errors

```
EINVAL             Invalid devhandle or msiqid or
                   uninitialized MSI EQ
```

21.4.10 pci_msi_getvalid

```
trap#              FAST_TRAP
function#          PCI_MSI_GETVALID
arg0               devhandle
arg1               msinum

ret0               status
ret1               msivalidstate
```

Return in *msivalidstate,* the current valid/enabled state for the MSI defined by the arguments *devhandle, msinum*.

21.4.10.1 Errors

```
EINVAL             Invalid devhandle or msinum
```

21.4.11 pci_msi_setvalid

```
trap#              FAST_TRAP
function#          PCI_MSI_SETVALID
arg0               devhandle
arg1               msinum
arg2               msivalidstate

ret0               status
```

Set the valid/enabled state of the MSI described by the arguments *devhandle, msinum* to the valid/enabled state defined by the argument *msivalidstate*

21.4.11.1 Errors

```
EINVAL                  Invalid devhandle, msinum or msivalidstate
```

21.4.12 pci_msi_getmsiq

```
trap#              FAST_TRAP
function#          PCI_MSI_GETMSIQ
arg0               devhandle
arg1               msinum

ret0               status
ret1               msiqid
```

For the MSI defined by the arguments *devhandle, msinum* return the MSI EQ that this MSI is bound to in the return value *msiqid*.

21.4.12.1 Errors

EINVAL	Invalid *devhandle* or *msinum* or msi unbound.

21.4.13 pci_msi_setmsiq

```
trap#              FAST_TRAP
function#          PCI_MSI_SETMSIQ
arg0               devhandle
arg1               msinum
arg2               msitype
arg3               msiqid

ret0               status
```

Set the target msiq of the MSI defined by the arguments *devhandle, msinum* to the MSI EQ id defined by the argument *msiqid*.

21.4.13.1 Errors

EINVAL	Invalid *devhandle*, *msinum* or *msiqid*

21.4.14 pci_msi_getstate

```
trap#              FAST_TRAP
function#          PCI_MSI_GETSTATE
arg0               devhandle
arg1               msinum

ret0               status
ret1               msistate
```

Return the state of the MSI defined by the arguments *devhandle, msinum*. If the MSI is not initialized, returns the state PCI_MSISTATE_IDLE.

21.4.14.1 Errors

EINVAL	Invalid *devhandle* or *msinum*

21.4.15 pci_msi_setstate

```
trap#               FAST_TRAP
function#           PCI_MSI_SETSTATE
arg0                devhandle
arg1                msinum
arg2                msistate

ret0                status
```

Set the state of the MSI defined by the arguments *devhandle, msinum* to the state defined by the argument *msistate*.

21.4.15.1 Errors

```
EINVAL              Invalid devhandle or msinum or msistate
```

21.4.16 pci_msg_getmsiq

```
trap#               FAST_TRAP
function#           PCI_MSG_GETMSIQ
arg0                devhandle
arg1                msgtype

ret0                status
ret1                msiqid
```

For the msg defined by the arguments *devhandle, msgtype* return the MSI EQ that this msg is bound to in the return value *msiqid*.

21.4.16.1 Errors

```
EINVAL              Invalid devhandle or msgtype.
```

21.4.17 pci_msg_setmsiq

```
trap#               FAST_TRAP
function#           PCI_MSG_SETMSIQ
arg0                devhandle
arg1                msg
arg2                msiqid

ret0                status
```

Set the target msiq of the msg defined by the arguments *devhandle, msgtype* to the MSI EQ id defined by the argument *msiqid*.

21.4.17.1 Errors

```
EINVAL              Invalid devhandle, msgtype or msiqid
```

21.4.18 pci_msg_getvalid

```
trap#              FAST_TRAP
function#          PCI_MSG_GETVALID
arg0               devhandle
arg1               msgtype

ret0               status
ret1               msgvalidstate
```

Return in *msgvalidstate,* the current valid/enabled state for the msg defined by the arguments *devhandle, msgtype*.

21.4.18.1 Errors

```
EINVAL             Invalid devhandle or msgtype
```

21.4.19 pci_msg_setvalid

```
trap#              FAST_TRAP
function#          PCI_MSG_SETVALID
arg0               devhandle
arg1               msgtype
arg2               msgvalidstate

ret0               status
```

Set the valid/enabled state of the msg described by the arguments *devhandle,* msg to the valid/enabled state defined by the argument *msgvalidstate*

21.4.19.1 Errors

```
EINVAL             Invalid devhandle, msgtype or msgvalidstate
```

22 UltraSPARC T1 performance counters

22.1 Introduction

An UltraSPARC T1 processor has one JBus, and four DRAM controllers integrated onto the same circuit. Each of these components contains counters that may be programmed to monitor and count specific events. A complete description of the UltraSPARC T1 performance counters is given in the UltraSPARC T1 Supplement to UltraSPARC Architecture 2005 manual.

Access the memory (DRAM) controller and JBus performance counters of a UltraSPARC T1 processor system is provided via an hypervisor API service. In a system configured with more than one guest domain, only one guest is allowed access to these performance counters. A machine description property ("perfctraccess") indicates that a guest is allowed access to the performance registers and this is enforced by the hypervisor.

22.2 Definitions

Each DRAM and JBus performance register is assigned a unique performance register (PerfReg) number for reading/writing purposes as follows:

PerfReg	Description
0	JBus Performance control register
1	JBus Performance counter register
2	DRAM Performance control register 0
3	DRAM Performance counter register 0
4	DRAM Performance control register 1
5	DRAM Performance counter register 1
6	DRAM Performance control register 2
7	DRAM Performance counter register 2
8	DRAM Performance control register 3
9	DRAM Performance counter register 3

22.3 API calls

22.3.1 niagara_get_perfreg

```
trap#               FAST_TRAP
function#           NIAGARA_GET_PERFREG
arg0                perfreg

ret0                status
ret1                value
```

This service reads the value of the DRAM/JBus performance register,as selected by the *perfreg* argument. Upon successful completion, it returns an EOK status and the performance register *value*.

22.3.1.1 Errors

```
EINVAL              Invalid performance register number
ENOACCESS           No access allowed to performance registers
```

22.3.2 niagara_set_perfreg

```
trap#               FAST_TRAP
function#           NIAGARA_SET_PERFREG
arg0                perfreg
arg1                value

ret0                status
```

This service sets the DRAM/JBus performance register, as specified by the *perfreg*, to *value*. Upon successful completion, it updates the specified performance register value and returns EOK status.

22.3.2.1 Errors:

```
EINVAL              Invalid performance register number
ENOACCESS           No access allowed to performance registers
```

23 Niagara-1 MMU statistics counters

23.1 Introduction

This section describes the hypervisor API to support MMU statistics collection on a Niagara-based system. This API is intended for UltraSPARC T1-specific performance measurement.

23.2 Hypervisor API for Niagara MMU statistics collection

On Niagara, hypervisor maintains MMU statistics. Privileged code provides Hypervisor a buffer wherein these statistics can be collected. After the successful configuration of the buffer, it is continuously updated (hits increased and ticks updated).

23.2.1 MMU statistic buffer format

The MMU statistics buffer has a fixed size, format and content as defined below:

offset (bytes)	size (bytes)	field
0x0	0x8	IMMU TSB hits ctx0, 8KByte TTE
0x8	0x8	IMMU TSB ticks ctx0, 8KByte TTE
0x10	0x8	IMMU TSB hits ctx0, 64KByte TTE
0x18	0x8	IMMU TSB ticks ctx0, 64KByte TTE
0x20	0x10	reserved
0x30	0x8	IMMU TSB hits ctx0, 4MByte TTE
0x38	0x8	IMMU TSB ticks ctx0, 4MByte TTE
0x40	0x10	reserved
0x50	0x8	IMMU TSB hits ctx0, 256MByte TTE
0x58	0x8	IMMU TSB ticks ctx0, 256MByte TTE
0x60	0x20	reserved
0x80	0x8	IMMU TSB hits ctxnon0, 8KByte TTE
0x88	0x8	IMMU TSB ticks ctxnon0, 8KByte TTE
0x90	0x8	IMMU TSB hits ctxnon0, 64KByte TTE
0x98	0x8	IMMU TSB ticks ctxnon0, 64KByte TTE
0xA0	0x10	reserved
0xB0	0x8	IMMU TSB hits ctxnon0, 4MByte TTE
0xB8	0x8	IMMU TSB ticks ctxnon0, 4MByte TTE
0xC0	0x10	reserved
0xD0	0x8	IMMU TSB hits ctxnon0, 256MByte TTE
0xD8	0x8	IMMU TSB ticks ctxnon0, 256MByte TTE
0xE0	0x20	reserved
0x100	0x8	DMMU TSB hits ctx0, 8KByte TTE
0x108	0x8	DMMU TSB ticks ctx0, 8KByte TTE
0x110	0x8	DMMU TSB hits ctx0, 64KByte TTE
0x118	0x8	DMMU TSB ticks ctx0, 64KByte TTE
0x120	0x10	reserved

offset (bytes)	size (bytes)	field
0x130	0x8	DMMU TSB hits ctx0, 4MByte TTE
0x138	0x8	DMMU TSB ticks ctx0, 4MByte TTE
0x140	0x10	reserved
0x150	0x8	DMMU TSB hits ctx0, 256MByte TTE
0x158	0x8	DMMU TSB ticks ctx0, 256MByte TTE
0x160	0x20	reserved
0x180	0x8	DMMU TSB hits ctxnon0, 8KByte TTE
0x188	0x8	DMMU TSB ticks ctxnon0, 8KByte TTE
0x190	0x8	DMMU TSB hits ctxnon0, 64KByte TTE
0x198	0x8	DMMU TSB ticks ctxnon0, 64KByte TTE
0x1A0	0x10	reserved
0x1B0	0x8	DMMU TSB hits ctxnon0, 4MByte TTE
0x1B8	0x8	DMMU TSB ticks ctxnon0, 4MByte TTE
0x1C0	0x10	reserved
0x1D0	0x8	DMMU TSB hits ctxnon0, 256MByte TTE
0x1D8	0x8	DMMU TSB ticks ctxnon0, 256MByte TTE
0x1E0	0x20	reserved

Note: "ticks" is the cumulative time spend handling the specified hit measured via deltas in the %tick register

23.3 API calls

23.3.1 niagara_mmustat_conf

```
trap#               FAST_TRAP
function#           NIAGARA_MMUSTAT_CONF
arg0                raddr

ret0                status
ret1                prev_raddr
```

This function enables MMU statistic collection and supplies the buffer to deposit the results for the current virtual CPU. The real address of the buffer, *raddr*, is supplied in arg0. The return value, ret1, is the previously specified buffer (*prev_raddr*), or zero for the first invocation.

If *raddr* is zero MMU statistic collection is disabled for the current virtual CPU and any previously supplied buffer is no longer accessed.

If an error is returned no statistics are collected (equivalent to passing an *raddr* of zero).

The initial contents of the buffer should be zero otherwise the collected statistics will be meaningless.

23.3.1.1 Errors

ENORADDR	Invalid *raddr*
EBADALIGN	*raddr* not aligned on 64-byte boundary
EBADTRAP	API not supported (all non-Niagara1 architectures)

23.3.2 niagara_mmustat_info

```
trap#               FAST_TRAP
function#           NIAGARA_MMUSTAT_INFO

ret0                status
ret1                raddr
```

This function provides an idempotent mechanism to query the state and real address of the currently configured buffer.

The real address of the current buffer, *raddr*, or zero, if no buffer is defined, is returned in ret1.

23.3.2.1 Errors

EBADTRAP	API not supported (all non-Niagara1 architectures)

24 Appendix A: How to use a machine description

A machine description ("MD") contains both explicit information about resources within a machine - detailed by specific nodes within the MD, and implicit information about the relationship of those resources - detailed by how nodes are interconnected into a relationship graph. We detail the relationship properties later in this section.

24.1 Using the MD as a list

For the simplest of sun4v guest operating environments, details of memory system hierarchy or even cache sizes are of little to no importance. Rather, basic information such as available memory regions and numbers of virtual CPUs are sufficient for the environment to function.

Therefore the MD is designed to enable the extraction of basic information without the need to parse any of the inter-relational information also provided.

For example, a simple guest may wish to simply determine the number of CPUs available in the machine. Within the MD each CPU is represented by a node of type "cpu" (please see section 8.9 for the definition of node types).

A guest may then, starting at the first node in the MD, simply linearly walk the list of nodes from one to the next in the list looking for nodes of a specific type. As each specific node is found properties may then be read from within that node. Pseudo code for this is illustrated in figure 5 below.

```
int find_node_idx(uint_t *bufferp, char *namep)
{
       struct MD_HEADER *hdrp;
       struct MD_ELEMENT * nodep;
       int i, nelems;
       char *strp;

       hdrp = (void*)bufferp;
       nodep = (void*) (bufferp+16);
       nelems = hdrp->node_blk_sz / 16;
       strp = buffer + 16 + hdrp->node_blk_sz;

       for (i=0; i<nelems; i=nodep[i].d.val) {
              char *sp;
              if (strcmp(strp+node[i].name_offset,
                     namep)==0) return i;
       }

       return (-1);  /* failed */
}
```

Figure 5Pseduo C-code for walking the list of nodes

24.2 Accelerating string lookups

To search for specific nodes or properties within a node, list element names need to be matched against known strings. The name for each list element is indirectly referenced in the name block of the machine description.

The basic method of searching for nodes or properties implies that for each tagged element in the machine description list, the name string must be found (using the offset in the element) and then the string compared against the desired string value.

While providing correct results these numerous string compares slow searching of the machine description.

The string match process may be short circuited due to the property of uniqueness of strings in the name block. The name block is constructed to guarantee that each string appears only once in the name block regardless of the number of times it is referenced by different elements. Since a desired string (e.g. “cpu”) can appear at most once in the name block, the index to that string in the name block becomes as unique as the string itself.

With this knowledge a more trivial method of searching the MD, is to first find the strings of interest in the name block - thus identifying the unique index for each string name. Then the MD itself can be seached by trivially matching the first 64 bytes of each element.

For example, suppose we wish to count the number of cpus represented in the MD. We first identify the string “cpu” in the name block; for our example it might appear at index 0x123. Thus any element uniquely identify the start of a cpu node will have the tag value 'N', name length of 4 (3 plus the nul string terminator) and name offset of 0x123. So then in the binary image of our example MD the first 64bits of any “cpu” node element will have the unique value of 0x4e0300000123.

A trivial linear search of the MD for this pattern enables nodes of type “cpu” to be counted;

Similarly, sought elements within a node can be matched using the same method of testing the first 64bits of the element structure.

Elements describing the start of a node have the specific property that the value field (elem_ptr->d.val) holds the index of the element for the next node in the machine description. So when searching specifically for node elements, other elements in the MD are trivially skipped thus speeding the search;

It is recommended that guests using the MD initially search and cache the indices of desired strings from the MD name block to avoid even the cost of finding the matching string index for each new MD search.

It should noted however, that the name block is unique to a particular MD. If the guest requests a new copy of a MD from the hypervisor, there is no guarantee that strings will have the same indices in the name block of the new MD as they have in the name block of the old MD.

24.3 Directed Acyclic Graph

The intrinsic Machine Description (MD) is a directed acyclic graph (DAG) of nodes describing resources or information available within a machine. This information is provided upon request to a guest operating system via the machine description request API.

24.3.1 Graph nodes

The DAG nodes are defined by the "NODE" element within the element list, and contain all the properties and arcs described until the subsequent NODE_END element. DAG node names form a well defined name space such that a particular name describes the type of a well defined entity. A different type of entity must be described by a node of a different name. For example, a CPU may be described by of type "cpu", while a cache is described by a node of type "cache".

Each node is a specific instance of the entity it describes. Properties or named values held within that node provide relevant details of the corresponding entity. For example, a cache node will hold a list of properties describing attributes of that cache.

As a node is defined by a specific "NODE" element within the element list, then for a specific MD, we can uniquely refer to that node by the index of its starting node element withing the element list. Thus if a "cpu" node starts at list element number 27, then a unique reference to that "cpu" node is the index value 27.

Using these index values for node start list elements, we can now provide pointers or "arcs" to point to other nodes. In the construction of the MD element list, we define the 64bit data payload of a "NODE" element to contain the index to the next "NODE" element in the element list. Thus a simple linear list of nodes is formed within the MD element list that enables searching for nodes of specific types without having to scan every list element looking for "NODE" and "NODE_END" tags.

Simlarly, using the PROP_ARC, type we can build a link or arc from one node to another. The value field of a PROP_ARC element is the 64bit element index of the "NODE" element pointed to. It is illegal for a PROP_ARC element to point to anything other than a NODE element, or a NOOP element (outside a node).

24.4 DAG construction

A DAG is constructed as described above by arcs that link the nodes together. The interconnection of these arcs explicitly defines the relationship between the nodes. For example, if node A has an arc to node C and node B has an arc to node C then the relationship exposed is that within the graph both nodes A and B share node C ***and*** any nodes that C arcs to. In the example illustration shown in the figure below we can see an instruction cache that is shared by two cpu nodes. The sharing is indicated by the existence of arcs from each cpu node to the same cache node.

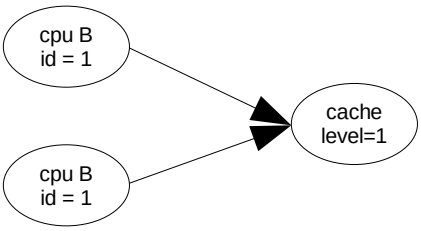


Figure 6Sharing relationship between nodes

The default DAG described within the MD is defined by arcs (element type PROP_ARC) with a name of "fwd". For convenience in walking this DAG, arcs named "back" are also provided that define the inverse DAG. Thus for every node A that has a "fwd" arc pointing to another node B, there is a corresponding "back" arc for node B pointing back to A.

The use of named arcs enables other DAGs to be built and contained within the same MD, however none other than the DAGs defined by the "fwd" and "back" arcs are currently defined.

24.5 Required nodes

The MD DAG will vary according to the resources available within a machine, and certain nodes may be present in a machine on one machine architecture, but not on a different machine architecture.

The MD concept is designed to allow for certain nodes to be "optional", however, to allow for the MD to be useable at all certain nodes must be defined and present in the description. These are "required" nodes and are guaranteed to be present if the resource they describe is present within the machine.

Nodes not defined in this specification must be ignored by system software.

24.6 The vanilla MD

Normally a MD is a full description of the resources available to specific logical domain. However, it is a requirement for any sun4v guest operating system that it be able to handle any machine description capable of being defined by this document and its subsequent revisions. To this end, a Guest operating system must be able to ignore / skip over nodes whose type and definitions the OS has never seen before, and most importantly that same Guest must follow some default fall-back behavior when information is not available.

To test the requirement for a default fall-back behavior, we define a "vanilla" description that contains only the core and required nodes for a given platform. This guarantees that a Guest OS is given no information about the platform upon which it is running, and to test that it continues to boot and execute - though optimal performance is no longer required.

The nodes in the vanilla MD are therefore required and sufficient to describe a guest environment for a basic sun4v compatible Operating System.

24.7 Formation and meaning of a DAG

As mentioned above a machine description currently contains only one DAG, and this is defined by all arcs with the name "fwd". As a courtesy, in order to speed certain searches, the MD also contains the inverse of this DAG built using arcs of name "back". Clearly the "back" DAG could be built by a guest from the "fwd" DAG, however the basic MD contains both to help lower the burden on the Guest.

Future revisions of this spec. may include new nodes, and importantly new DAGs within the same MD. Current software should be designed to ignore arcs with names other than "fwd" and "back" in order to remain future proof. Future MD will be implemented so as not to have conflicts with the vanilla fwd and back DAGs.

To understand how to use the DAGs in a MD consider the DAG built using the "fwd" arcs.

The root of the "fwd" DAG is a node of type "root". This is by definition the very first node in the MD. It can be found very simply by scanning the MD element list for the first NODE definition (though unfortunately, due to the existence of NOOP elements, this need not be at element index 0).

From the root node, "fwd" arcs lead to nodes describing the various components within the logical domain a guest is using.

The root node in turn contains “fwd” arcs to collective nodes for cpus, memory and various forms of I/O, as well as nodes targeted to specific consumers such as OpenBoot.

25 Appendix B: Number Registry

This appendix provides a registry of API services, their assigned trap and function numbers, and currently defined version groups and version numbers.

The definitions of the API groupings for the versioning API (§9) are as follows:

Group	Number (api_group)	Group Definition
Common	0x0	sun4v platform
	0x1	core APIs
Technology	0x100	PCI
	0x101	Logical Domain Channels
	0x102	Service Channels (*)
Performance measurement	0x200	UltraSPARC T1 performance counters

(*) These calls have now been deprecated, and are described only for compatibility with old platform firmware.

25.1 Hyper-fast Trap numbers

For hyper-fast traps, the *sw_trap_numbers are* encoded in the Tcc instruction that enters the hypervisor:

Un-assigned trap numbers result in EBADTRAP being returned in %o0 as described in section 2.3.

25.2 FAST_TRAP Function numbers

Function numbers for fast-traps are provided in %o5 as a 64-bit value.

Un-assigned function numbers used for fast-traps result in EBADTRAP being returned in %o0 as described in section 2.3.

25.3 CORE_TRAP Function numbers

CORE_TRAP APIs are defined and guaranteed present for all sun4v hypervisor versions. These APIs follow the same calling conventions as FAST_TRAP API services. Four CORE_TRAP functions are currently defined as follows;

API_VERSION	defined in section 9.1.1.
API_PUTCHAR	an alias for FAST_TRAP function CONS_PUTCHAR .
API_EXIT	an alias for FAST_TRAP function MACH_EXIT.
API_GET_VERSION	defined in section 9.1.2.

25.4 Summary of API service trap and function numbers

Trap#	Func#	Versioning Group#	Vers#	Name	Defined in section
0x80	--	N/A	N/A	FAST_TRAP	-
0x83	--	0x001	1.0	MMU_MAP_ADDR	12.7.6
0x84	--	0x001	1.0	MMU_UNMAP_ADDR	12.7.8
0x85	--	0x001	1.0	TTRACE_ADDENTRY	18.3.5
0xff	--	N/A	N/A	CORE_TRAP	-
0x80	0x00	0x001	1.0	MACH_EXIT	10.1.1
0x80	0x01	0x001	1.0	MACH_DESC	10.1.2
0x80	0x02	0x001	1.0	MACH_SIR	10.1.3
0x80	0x03	0x001	1.1 *	MACH_SET_SOFT_STATE	10.1.4
0x80	0x04	0x001	1.1 *	MACH_GET_SOFT_STATE	10.1.5
0x80	0x05	0x001	1.1 *	MACH_SET_WATCHDOG	10.1.6
0x80	0x10	0x001	1.0	CPU_START	11.2.1
0x80	0x11	0x001	1.1 *	CPU_STOP	11.2.2
0x80	0x12	0x001	1.0	CPU_YIELD	11.2.5
0x80	0x14	0x001	1.0	CPU_QCONF	11.2.6
0x80	0x15	0x001	1.0	CPU_QINFO	11.2.7
0x80	0x16	0x001	1.0	CPU_MYID	11.2.9
0x80	0x17	0x001	1.0	CPU_STATE	11.2.10
0x80	0x18	0x001	1.0	CPU_SET_RTBA	11.2.3
0x80	0x19	0x001	1.0	CPU_GET_RTBA	11.2.4
0x80	0x20	0x001	1.0	MMU_TSB_CTX0	12.7.1
0x80	0x21	0x001	1.0	MMU_TSB_CTXNON0	12.7.2
0x80	0x22	0x001	1.0	MMU_DEMAP_PAGE	12.7.3
0x80	0x23	0x001	1.0	MMU_DEMAP_CTX	12.7.4
0x80	0x24	0x001	1.0	MMU_DEMAP_ALL	12.7.5
0x80	0x25	0x001	1.0	MMU_MAP_PERM_ADDR	12.7.7
0x80	0x26	0x001	1.0	MMU_FAULT_AREA_CONF	12.7.10
0x80	0x27	0x001	1.0	MMU_ENABLE	12.7.11
0x80	0x28	0x001	1.0	MMU_UNMAP_PERM_ADDR	12.7.9
0x80	0x29	0x001	1.0	MMU_TSB_CTX0_INFO	12.7.12
0x80	0x2a	0x001	1.0	MMU_TSB_CTXNON0_INFO	12.7.13
0x80	0x2b	0x001	1.0	MMU_FAULT_AREA_INFO	12.7.14
0x80	0x31	0x001	1.0	MEM_SCRUB	13.1.1
0x80	0x32	0x001	1.0	MEM_SYNC	13.1.2
0x80	0x42	0x001	1.0	CPU_MONDO_SEND	11.2.8
0x80	0x50	0x001	1.0	TOD_GET	15.1.1
0x80	0x51	0x001	1.0	TOD_SET	15.1.2
0x80	0x60	0x001	1.0	CONS_GETCHAR	16.1.1
0x80	0x61	0x001	1.0	CONS_PUTCHAR	16.1.2

Trap#	Func#	Versioning Group#	Vers#	Name	Defined in section
0x80	0x80	0x102	1.0	SVC_SEND	(*)
0x80	0x81	0x102	1.0	SVC_RCV	(*)
0x80	0x82	0x102	1.0	SVC_GETSTATUS	(*)
0x80	0x83	0x102	1.0	SVC_SETSTATUS	(*)
0x80	0x84	0x102	1.0	SVC_CLRSTATUS	(*)
0x80	0x90	0x001	1.0	TTRACE_BUF_CONF	18.3.1
0x80	0x91	0x001	1.0	TTRACE_BUF_INFO	18.3.2
0x80	0x92	0x001	1.0	TTRACE_ENABLE	18.3.3
0x80	0x93	0x001	1.0	TTRACE_FREEZE	18.3.4
0x80	0x94	0x001	1.0	DUMP_BUF_UPDATE	17.1.1
0x80	0x95	0x001	1.0	DUMP_BUF_INFO	17.1.2
0x80	0xa0	0x001	1.0	INTR_DEVINO2SYSINO	14.2.1
0x80	0xa1	0x001	1.0	INTR_GETENABLED	14.2.2
0x80	0xa2	0x001	1.0	INTR_SETENABLED	14.2.3
0x80	0xa3	0x001	1.0	INTR_GETSTATE	14.2.4
0x80	0xa4	0x001	1.0	INTR_SETSTATE	14.2.5
0x80	0xa5	0x001	1.0	INTR_GETTARGET	14.2.6
0x80	0xa6	0x001	1.0	INTR_SETTARGET	14.2.7
0x80	0xb0	0x100	1.0	PCI_IOMMU_MAP	20.4.1
0x80	0xb1	0x100	1.0	PCI_IOMMU_DEMAP	20.4.2
0x80	0xb2	0x100	1.0	PCI_IOMMU_GETMAP	20.4.3
0x80	0xb3	0x100	1.0	PCI_IOMMU_GETBYPASS	20.4.4
0x80	0xb4	0x100	1.0	PCI_CONFIG_GET	20.4.5
0x80	0xb5	0x100	1.0	PCI_CONFIG_PUT	20.4.6
0x80	0xb6	0x100	1.0	PCI_PEEK	20.4.7
0x80	0xb7	0x100	1.0	PCI_POKE	20.4.8
0x80	0xb8	0x100	1.0	PCI_DMA_SYNC	20.4.9
0x80	0xc0	0x100	1.0	PCI_MSIQ_CONF	21.4.1
0x80	0xc1	0x100	1.0	PCI_MSIQ_INFO	21.4.2
0x80	0xc2	0x100	1.0	PCI_MSIQ_GETVALID	21.4.3
0x80	0xc3	0x100	1.0	PCI_MSIQ_SETVALID	21.4.4
0x80	0xc4	0x100	1.0	PCI_MSIQ_GETSTATE	21.4.5
0x80	0xc5	0x100	1.0	PCI_MSIQ_SETSTATE	21.4.6
0x80	0xc6	0x100	1.0	PCI_MSIQ_GETHEAD	21.4.7
0x80	0xc7	0x100	1.0	PCI_MSIQ_SETHEAD	21.4.8
0x80	0xc8	0x100	1.0	PCI_MSIQ_GETTAIL	21.4.9
0x80	0xc9	0x100	1.0	PCI_MSI_GETVALID	21.4.10
0x80	0xca	0x100	1.0	PCI_MSI_SETVALID	21.4.11
0x80	0xcb	0x100	1.0	PCI_MSI_GETMSIQ	21.4.12

Trap#	Func#	Versioning Group#	Vers#	Name	Defined in section
0x80	0xcc	0x100	1.0	PCI_MSI_SETMSIQ	21.4.13
0x80	0xcd	0x100	1.0	PCI_MSI_GETSTATE	21.4.14
0x80	0xce	0x100	1.0	PCI_MSI_SETSTATE	21.4.15
0x80	0xd0	0x100	1.0	PCI_MSG_GETMSIQ	21.4.16
0x80	0xd1	0x100	1.0	PCI_MSG_SETMSIQ	21.4.17
0x80	0xd2	0x100	1.0	PCI_MSG_GETVALID	21.4.18
0x80	0xd3	0x100	1.0	PCI_MSG_SETVALID	21.4.19
0x80	0xe0	0x101	1.0	LDC_TX_QCONF	19.4.1
0x80	0xe1	0x101	1.0	LDC_TX_QINFO	19.4.2
0x80	0xe2	0x101	1.0	LDC_TX_GET_STATE	19.4.3
0x80	0xe3	0x101	1.0	LDC_TX_SET_QTAIL	19.4.4
0x80	0xe4	0x101	1.0	LDC_RX_QCONF	19.4.5
0x80	0xe5	0x101	1.0	LDC_RX_QINFO	19.4.6
0x80	0xe6	0x101	1.0	LDC_RX_GET_STATE	19.4.7
0x80	0xe7	0x101	1.0	LDC_RX_SET_QHEAD	19.4.7
0x80	0x100	0x200	1.0	NIAGARA_GET_PERFREG	22.3.1
0x80	0x101	0x200	1.0	NIAGARA_SET_PERFREG	22.3.2
0x80	0x102	0x200	1.0	NIAGARA_MMUSTAT_CONF	23.3.1
0x80	0x103	0x200	1.0	NIAGARA_MMUSTAT_INFO	23.3.2
0xff	0x00	N/A	N/A	API_SET_VERSION	9.1.1
0xff	0x01	N/A	N/A	API_PUTCHAR	16.1.2
0xff	0x02	N/A	N/A	API_EXIT	10.1.1
0xff	0x03	N/A	N/A	API_GET_VERSION	9.1.2

* These version numbers are provisional

25.5 Error codes

When a hypervisor API returns, unless explicitly described by the API service, the 64-bit value in %o0 will be one of the following error identification values.

Value	Mnemonic	Comment
0	EOK	Successful return
1	ENOCPU	Invalid CPU id
2	ENORADDR	Invalid real address
3	ENOINTR	Invalid interrupt id
4	EBADPGSZ	Invalid pagesize encoding
5	EBADTSB	Invalid TSB description
6	EINVAL	Invalid argument
7	EBADTRAP	Invalid function number
8	EBADALIGN	Invalid address alignment
9	EWOULDBLOCK	Cannot complete operation without blocking
10	ENOACCESS	No access to specified resource
11	EIO	I/O Error
12	ECPUERROR	CPU is in error state
13	ENOTSUPPORTED	Function not supported
14	ENOMAP	No mapping found
15	ETOOMANY	Too many items specified / limit reached
16	ECHANNEL	Invalid LDC channel