UltraSPARC Virtual Machine Specification

UltraSPARC Virtual Machine Specification

Compiled from hg version 3033604f0239 3.0-draft7

Publication date 2012-03-13 19:43

Table of Contents

List of Figures

List of Tables

Preface

1. Foreward

This document is the software specification for the UltraSPARC virtual machine environment. The virtual machine environment is created by 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 an operating system, but it also enables the separation of physical resources—this allows multiple virtual machines to be hosted on a single platform. Each virtual machine is its own self-contained partition (or “Logical Domain”) capable of supporting an independent operating system image.

This document details the UltraSPARC virtual machine environment together with the calling conventions and detailed specifications of the virtual machine interfaces provided to a Logical Domain.

This document is intended for operating system and firmware engineers looking for detailed information on the UltraSPARC virtual machine environment, as well as the merely curious.

2. Related specifications

The UltraSPARC 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 form the basis of the hardware architecture. This architecture incorporates the Level-1 SPARC v9[sparcv9] 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. A full description of available machine registers is given in the UltraSPARC Architecture[ua2007].

In addition to the UltraSPARC Architecture manual, processor specific details for each UltraSPARC processor are provided in the supplemental manuals corresponding to each chip. These manuals provide information on chip specific hardware details, such as performance counters.

At the time of writing the latest versions of these specifications 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.

Chapter 1. Overview

This document provides the detailed interface specifications for the UltraSPARC virtual machine environment. However, before the deep dive into the technical details, this section aims to provide an overview of the entire architecture: the intentions behind much of the design, the individual components and how they operate.

1.1. Architectural requirements

We start with the foundation stone for the UltraSPARC virtual machine environment; the UltraSPARC Hypervisor.

The fundamental need to support multiple concurrently running operating systems on the same platform was the goal. However, the UltraSPARC Hypervisor had to meet four architectural requirements in achieving this; security, heterogeneity, availability, and of course high performance.

One of the significant value propositions of a virtualization solution is the ability to consolidate multiple workloads onto a single platform and thereby increase the overall efficiency of a datacenter. Achieving this efficiency is however a non-trivial problem, after all operating systems have been able to run multiple applications concurrently for decades, and yet datacenter administrators have traditionally avoided doing so. Why?

In practice deploying an application in a datacenter often involves the careful selection and testing of a specific operating system together with its requisite patches and tuning parameters. Once selected, upgrades to that application or even the underlying OS often occur on a timetable related to the application vendors releases. Consequently, in an environment with multiple applications it is difficult to find OS versions, patches etc. that work well for all applications, and for the same reasons upgrades have to be carefully coordinated. So, it's usually just easier from an administrative perspective to assign a unique machine to a specific application / task.

To deploy a OS virtualization solution into a typical data center environment for use as a consolidation tool, the Hypervisor must be capable of supporting multiple different (heterogeneous) operating systems.

Often it is the case that different applications are owned and run by different departments within a corporation, or even different external customers. Consider a buggy or even malicious OS patch installed in an operating system— while that could spell disaster for that specific virtual machine it should still be effectively isolated from other virtual machines on the same platform. This means that an effective virtual machine solution must provide strong security between virtual machines. Weak security (e.g. a poorly chosen password) within one virtual machine should not leave the rest of the machine vulnerable to attack either directly or by denial of service.

Similarly, placing so many eggs in one basket raises the need for improved fault tolerance, and increased availability in the event of a failure. No matter what the capability for fault handling of an individual OS, it is only as effective as the underlying hypervisor's ability to report and manage faults upwards. For example, if a failing CPU can take out the entire hypervisor, the fault is not limited simply to the virtual machine using that resource but is now expanded to the entire machine. Clearly then an effective availability capability is required from a hypervisor in the event of system component failures.

Quite simply; the goal for a hypervisor is to create virtual machines that have the attributes of classic independent machines, but consolidated onto a single platform where resources can be shared and for that sharing to be as efficient as possible so that the overheads do not overwhelm the overall benefit of consolidation.

The complete Logical Domaining solution is designed to behave as much like a collection of independent machines as possible, even to the extent of all of the virtual machines being able to boot, shutdown, crash

and reboot independently of each other. The remainder of this section describes the final architectural implementation to meet these requirements.

1.2. The hypervisor and sun4v architecture

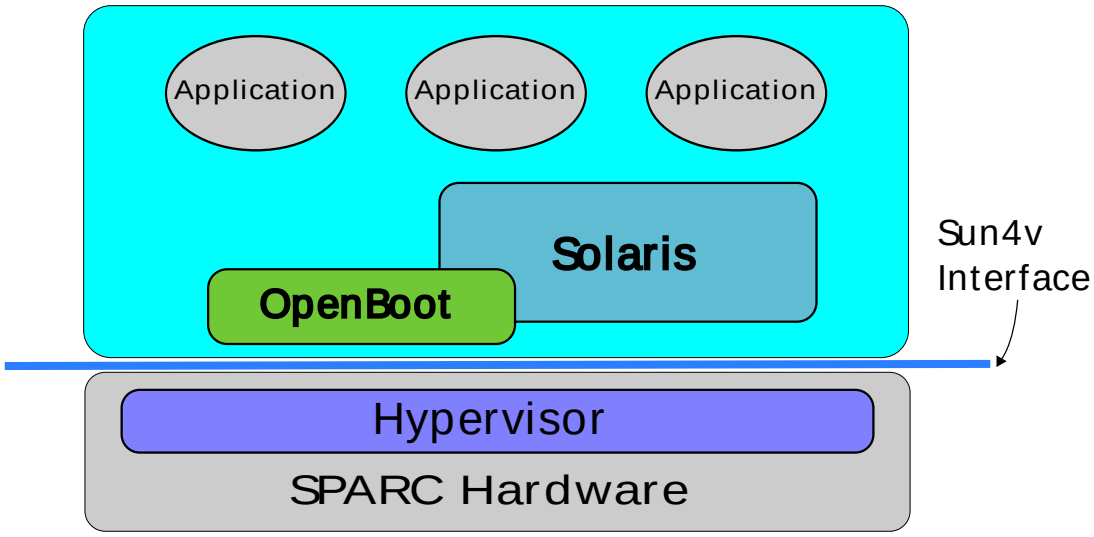
Unlike other hypervisor solutions, the UltraSPARC Hypervisor is not booted from disk like a traditional operating system. Instead the UltraSPARC Hypervisor is architected to be integrated into the firmware PROM of each hardware platform, and starts up immediately after system initialization. This approach enables chip-set specific code to be delivered directly with each platform as it is released. No careful patching or tuning is required because each hypervisor is delivered with, and is specific to, a particular platform.

The traditional firmware boot loader for SPARC, OpenBoot, is completely virtualized and each logical domain uses its own independent copy for booting.

Key to high performance, as well as minimizing bugs and problems in the field, is keeping the hypervisor as simple as possible. The overall Logical Domain architecture reflects the desire to keep features out of the hypervisor and seat them within the virtual machines themselves. Quite simply, fewer lines of hypervisor code mean fewer bugs, and a greater test coverage before each platform release.

The result then is a hypervisor that provides support functions to guest operating systems via a well defined and stable software interface. Coupled with hardware support for protection and isolation the resultant virtual machine environment is called the “sun4v” architecture.

Figure 1.1. Sun4v Architecture



The figure above illustrates the sun4v interface provided by the UltraSPARC Hypervisor, and its relationship to the virtualized clients that run within a logical domain.

1.3. Privilege, isolation and virtualization

In order to provide isolation and protection the UltraSPARC execution model is extended with a hyper-privileged mode. This additional privilege level is for the hypervisor alone, leaving guest operating systems and their applications running in more restrictive modes that deny access to sensitive control registers and memory. The hypervisor in turn abstracts underlying hardware resources and exposes a subset to each virtual machine or “Logical Domain”.

Consequently, guest operating systems within Logical Domains can only access or control platform resources explicitly made available by the hypervisor. Typically that access is provided via hypervisor API calls made by a guest operating system, where the parameters can be checked and approved by the hypervisor prior to being acted upon (or rejected). In a few cases where higher performance is required (such as interrupt or timer handling) hardware support provides specific registers accessible from within a virtual machine.

All sun4v architected registers are defined to be idempotent, and hypervisor API interfaces set or clear state explicitly rather than by side effect. These criteria enable the complete state of a virtual machine to be unloaded from machine resources, encapsulated, and then later resurrected on different hardware resources— even on a different physical machine. This fundamental capability allows a hypervisor to support a range of useful feature. For example, simple capabilities such as the time-multiplexing of multiple virtual CPUs onto a single physical CPU, or more complex functionality such as the live-migration of a running virtual machine from one physical platform to another.

With hardware support the hypervisor also virtualizes memory. Physical memory is subdivided and allocated to different domains. A unique address space is created for each virtual machine and supported by the hypervisor.

By not being able to address anything outside its own domain a virtual machine is rigorously isolated from memory and memory mapped devices it does not own. Hardware tags in the CPU translation look aside buffers (TLBs) strictly enforce the separation of these address spaces allowing multiple virtual CPUs to be efficiently time multiplexed onto a single physical CPU.

1.4. Direct I/O

While the hypervisor provides APIs for basic system components such as virtual CPUs, more complex I/O devices are handled differently.

Most modern I/O devices designed for performance have fairly sophisticated device drivers to handle multiple functions, concurrency, complex bug workarounds and even to upload device specific firmware. Moreover, these I/O devices are often provided by third party vendors that have developed their own closed-source device drivers.

Consequently, the UltraSPARC Hypervisor makes no attempt to virtualize hardware I/O devices. Instead I/O devices are directly mapped into Logical Domains.

This approach is enormously beneficial on three levels;

Firstly, by avoiding a loadable device driver model, there are no possible security holes by which a guest OS or an operator can insert buggy or malicious code into the hypervisor.

Secondly, no device specific patching or tuning of the hypervisor is required. This better matches the stability model expected of system firmware. This is particularly important given that many I/O devices are plug in cards from third party vendors and true testing can therefore only be achieved in the field.

Thirdly, no loadable device driver capability means no need for a device driver framework. This significantly reduces the size of the hypervisor and therefore the number and range of possible bugs. For example, at the time of writing, the Solaris device driver interface (DDI) framework contains more lines of code than the entire hypervisor source base for either the UltraSPARC-T1 or UltraSPARC-T2 processors.

Depending on the platform hardware capabilities, devices can be mapped into individual domains at the system bus level, the device level, or even down to functions within devices. (The latter requires capable multifunction devices such as the UltraSPARC-T2's network interface unit, or PCI-IOV devices). To

achieve this the hypervisor relies on the capabilities of the processor's memory management unit (MMU) to control CPU access to device registers. Similarly, it requires the use of an I/O MMU to map and control device access to system memory. Specifically, the I/O MMU is used to prevent one domain being able to DMA to or from memory belonging to other domains sharing the same system.

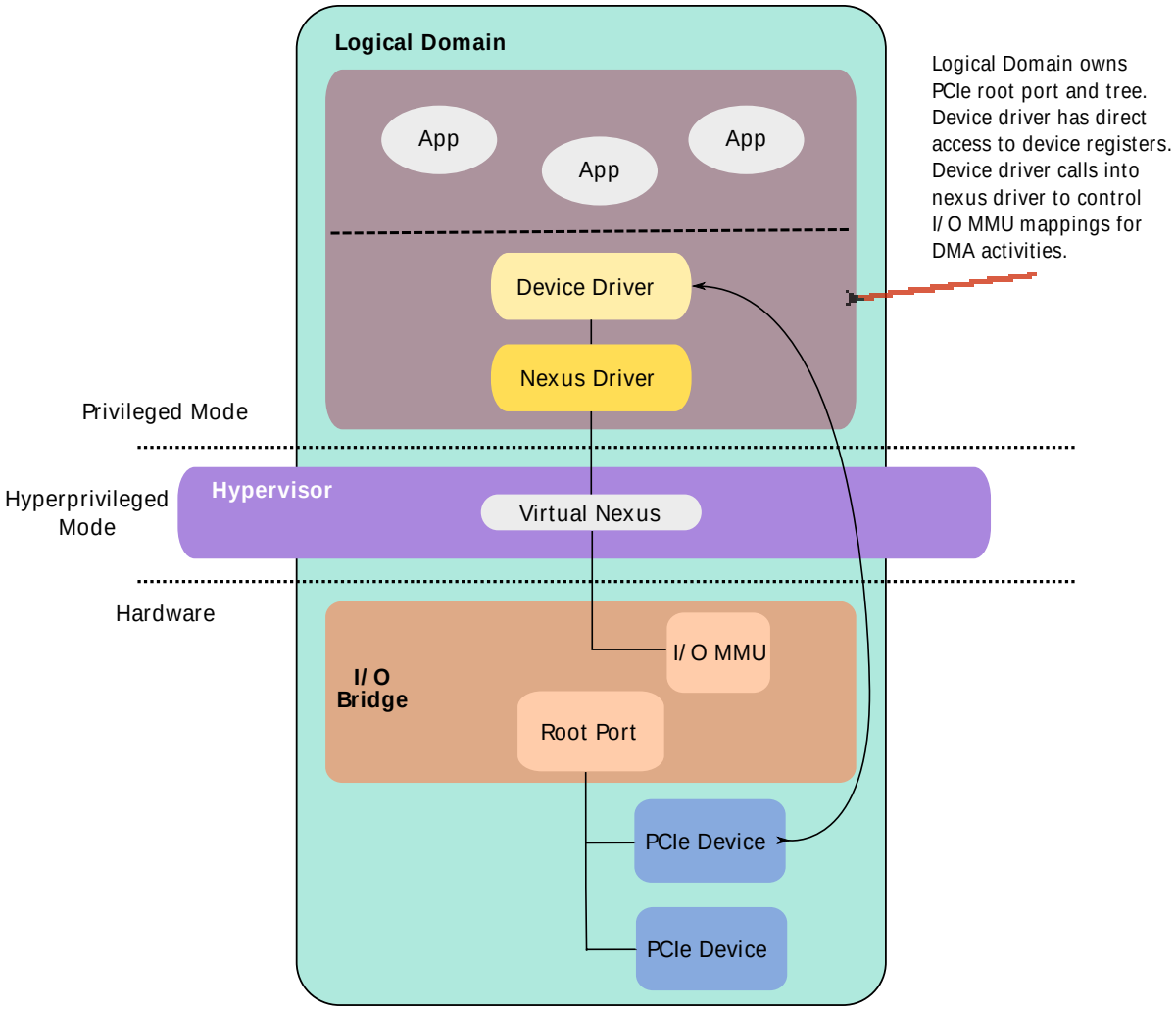
The result is performance I/O devices being exclusively assigned to specific logical domains. The guest operating systems running in those domains have direct access to the device registers and can configure device operations such as DMA activity. DMA mappings for the I/O MMU are configured using hypervisor APIs so that addresses specified can be validated by the hypervisor.

Device interrupts and system bus errors are directed from I/O devices via the hypervisor to virtual CPUs in a virtualized form. This enables the hypervisor to remap, suspend or context switch virtual CPUs on physical CPUs without risk of device interrupts being lost.

Thus domains with I/O devices can have direct control over those devices when performance is required. Furthermore, guest operating systems do not require special device drivers to run in logical domains and can continue to use the device driver frameworks they already have. This even allows legacy devices and drivers from non virtualized systems to be used. And finally, system administrators do not have to change their procedures when testing and patching of their operating systems to deal with third-party devices.

This model is the basic building block for all I/O in a Logical Domain system. However, it is insufficient when there are more logical domains than physical I/O devices available for use. To overcome this restriction, the architecture requires that some of the logical domains that are assigned physical I/O devices act as proxies on behalf of the other domains. For this to work the domains need to be able to communicate.

Figure 1.2. Direct I/O



1.5. Logical Domain Channels

A logical domain channel (LDC) is a point-to-point, full-duplex virtual link created between domains by the hypervisor. LDCs provide a data path, a means to share memory and a mechanism to deliver asynchronous interrupts between domains.

The most basic communication mechanism is the delivery of short (64-byte) datagrams along a logical domain channel. Guest operating system code can build higher level protocols for larger packet and reliable communication. Thus the complexity for sophisticated domain- to-domain protocols remains with each guest operating system, leaving the hypervisor to implement only the most basic transport mechanism.

In addition to the short message delivery capability, one domain can export memory to another directly for sharing. With a direct shared memory interface both domains can then communicate as fast as the implemented protocol and memory subsystem bandwidth will allow. Direct I/O devices can be configured to DMA directly to/from memory imported from or exported to another domain. Either domain can revoke the shared memory mapping at any time, and domains can only access the memory of another domain that has been explicitly exported to them.

1.5.1. Stateless connections

Logical Domain Channels may be closed by either domain, or by the hypervisor at any time. It is expected that guest operating systems utilizing LDCs are able to handle the arbitrary closure and re-connection of an LDC. After an LDC closes, if the connection comes up again, a guest operating system must re-negotiate the communication protocol without assumptions about the domain on the other side of the link.

This requirement is by convention and not enforced by the hypervisor, however it is specified in order to support the dynamic re-configuration of system services wherein a domain's LDC may be disconnected from one service and re-connected to a different service. A prime example of this is in the case of live-migration, where a domain is moved from one box to another and subsequently connected to new support domains on the new box.

Therefore domains utilizing LDC connections must be able to recover from a reset (closed and opened) connection by re-negotiating protocol interfaces and be able to re-submit any pending transactions.

1.5.2. LDC security

Security is a paramount concern with any communication mechanism. In particular, the problems traditionally associated with networks of machines have been architected out - namely; information leakage, authentication, faked credentials and denial of service attacks.

Unlike more general purpose communication mechanisms such as the Internet Protocol (IP), the hypervisor Logical Domain Channel APIs provide no capability for a domain to open a connection to another domain. LDCs can only be created by the system “Domain manager” (which we discuss later). Without a means to establish their own connections, domains do not have to deal with problems of addressing, connection management and authentication. A rogue domain cannot randomly connect to another domain. There is no mechanism by which to undertake activities like port scanning.

If a LDC exists between two domains it had to have been created by the administrator for a specific named purpose (for example a virtual disk interface), and both sides of that connection are clearly informed of the role they are expected to play. As LDCs are simple point-to-point connections there is no risk of information leakage to other domains via snooping techniques. Denial of service attacks are easily closed off by a recipient domain by simply ignoring the rogue LDC; traffic from other domains cannot be blocked since it arrives by separate point-to-point LDCs.

This unconventional approach to interconnecting domains is made possible because the virtual machines all execute on the same shared memory system. Higher level protocols such as TCP/IP are expected for use between applications in different domains or for in and out of box communication.

1.6. Machine Descriptions

Operating system code running within a virtual machine environment needs a means to discover the resources that are available within that environment. On traditional non-virtual machines hardware resources are typically probed for, which is an exhaustive process of testing hardware registers and waiting for lack of response or bus errors to indicate that suspected hardware is not in fact present.

In a para-virtualized world, there is no need to go through this arcane process to discover available resources. A simple hypervisor API provides a detailed description of the resources within the virtual machine. This description is called a “Machine Description” (or “MD”) and is a one-stop catalog of every resource a guest operating system has available.

Aside from the basics such as CPU and memory map details, a domain's MD also contains detailed relational information about resources, such as NUMA latencies and the sharing between caches in the memory system hierarchy.

Some of the information provided in the MD is mandatory, and the rest is typically advisory to be used for performance optimizations. The key advantage of storing this information with the hypervisor is that it is always retrievable by a domain. This avoids any bottle-necking on a “master” domain to disseminate the information. This allows for a simultaneous parallel boot after power-on of all logical domains in a system without the single-point-of-failure that a master domain would introduce into the boot process.

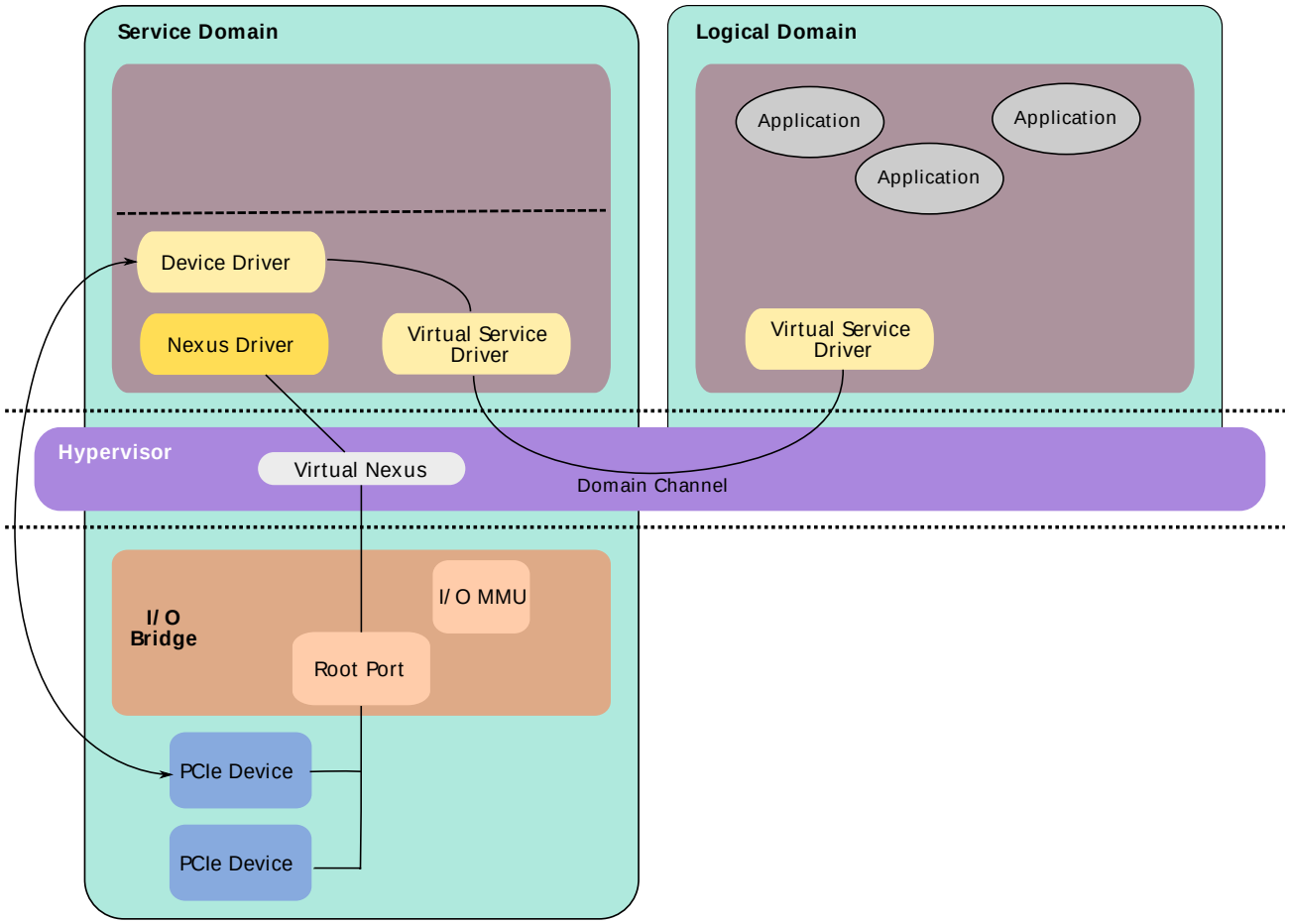
1.7. Virtual I/O

Direct I/O is the foundation model for I/O access in a Logical Domain system. However, it is possible to create more domains than there are physical I/O devices. In order to support sharing of I/O devices for virtualization, we enable some domains with direct I/O access to act as proxies on behalf of other domains.

Communication between client and proxy domains is achieved using a high level negotiated protocol over a dedicated LDC between domains. This document details the protocols currently in use between domains for proxy services such as disk and networking I/O. These protocols are not in anyway enforced by the hypervisor, and are a convention between domains.

As illustrated below a domain acting as a proxy for I/O is assigned a physical I/O device for direct access. For this reason it is defined as an “IO domain”. The domain runs the appropriate device driver for the specific hardware device.

Figure 1.3. Virtual I/O



The domain also runs a proxy service responsible for exporting an abstracted form of the device to other client domains. For this reason it is also designated as a “Service domain”.

Note: There is no requirement for a “service domain” to also be an “IO domain”. For example, a service domain may provide a virtual network switch to other clients without requiring a physical connection outside of the box; thus the domain may have no IO domain capability. (We will cover domain roles later in this section).

The proxy service run by the service domain receives I/O requests from each of its client domains, and is responsible for servicing those requests on behalf of its client. For example, a disk read request is sent to the service domain from its client. The LDC framework delivers the request to the service proxy in the service domain, the proxy is then responsible for deciphering the request and utilizing an IO device driver to schedule the request as appropriate. Once complete the proxy service then acknowledges completion back to the client domain.

1.7.1. Abstraction

Typically the communication protocol for a given service is highly abstracted and agnostic with regard to the physical device in the actual IO domain. For example, the disk server deployed with Solaris 10 uses internal Solaris interfaces to access an underlying storage medium on behalf of its clients. This enables storage bits to be provide by many different sources such as a disk, or a file or a RAID-ed volume — all of which is abstracted and invisible to the client at the other end of the LDC channel. The client instead sees is a disk service capable of reading and writing disk blocks according to the abstracted protocol.

This abstraction has enormous advantages when deploying multiple domains on a single system. For example, instead of a dedicated boot disk for each virtual machine, a simple file on a RAID protected filesystem can be used on the IO domain to store the boot disk images of each of the client domains. Backup of the domains then becomes a simple matter of backing up the files on the IO domain.

1.7.2. Stateless connections & multipathed I/O

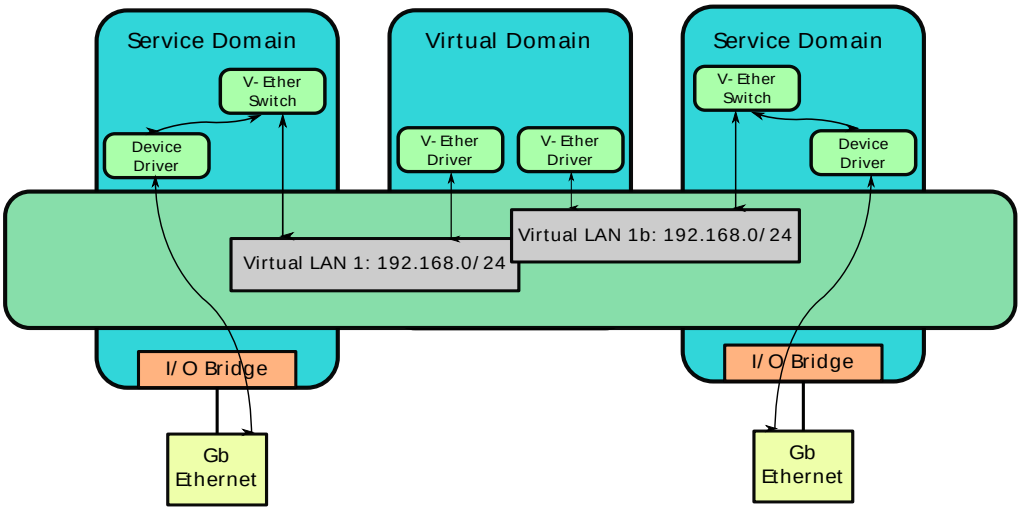
Following the requirements of the underlying LDC infrastructure, all virtual device protocols should be designed to be stateless or transactional. This allows for a LDC channel to be arbitrarily broken and reconnected (possibly to a different IO service). This functionality is expected by the domain manager, and is relied upon to support resiliency, live migration and dynamic fail-over of system services.

For example, in the basic proxy configuration illustrated above, if the service were a virtual disk service, and the service domain were to panic, (perhaps due to a hardware fault or buggy driver), the client domain would observe the LDC being reset and wait until the channel came back up again. Once the service has reestablished itself on the LDC, the client could re-negotiate the service capabilities and re-submit any uncompleted transactions.

The subtlety here is that the client domain was unaffected by the failure and rebooting of its service domain. This mirrors how the client and server model would function if provided on two physically separate machines. Another way to view this is as a means to harden IO devices and drivers that are prone to catastrophic failure by restricting them to their own domain.

In a more complex configuration, a multipath arrangement of I/O domains could be provided as illustrated in the figure below.

Figure 1.4. Virtual Network Multipath



In this example, a client domain (2) has access to an external network via two service IO domains. Assuming the client domain's operating system can support this kind of multipathed I/O, the configuration allows for the failure of almost any part of the system up to the client domain itself. Traffic can be easily diverted via the service domain on the right if the hardware (or operating system) in the service domain on the left should fail. In addition, because the protocol is designed as stateless, a service action (e.g. card swap or reboot) could bring the domain on the left back on line again, after which traffic from the redundantly connected client can be load balanced back.

It is easy to see with this infrastructure, how even scheduled outages can be avoided. For example, because the protocols are re-negotiated, a rolling service domain upgrade could be implemented first by upgrading and rebooting first the left and then the right service domain without loss of external connectivity.

This simple set of requirements, implemented by the virtual device protocols, allows for some very ingenious and robust virtual I/O infrastructures to be created. Thus not only can physical I/O devices be shared by multiple domains, but greater robustness and flexibility can be achieved using this kind of virtualization.

1.7.3. Virtual disk services

The virtual disk protocol defined in this specification assumes the stateless behavior described above. A straightforward LDC is created between the service domain and the client domain by the domain manager. The disk proxy service then slavishly responds to requests from the client.

In the same way that a service domain can support multiple client domains, a client domain can be configured to utilize multiple service domains. If supported by the client domain's operating system, this multipathed configuration can be used for redundant access to storage as described above.

1.7.4. Scalable virtual networking services

The virtual networking protocol defined in this specification allows for a full layer-2 Ethernet network switch to be created in a service domain. This switch can also be configured with multiple upstream ports (utilizing direct IO interfaces), or be configured to communicate with the local kernel for higher level routing or firewalling functionality.

Each of the client domains of the networking switch has a LDC to the service domain. In the most basic incarnation, network packets are sent by a client domain to its virtual network switch, that then forwards those packets to the appropriate destination. The destination may be upstream or simply another domain in the same machine.

The switch service protocol also provides broadcast and multicast functionality, as well as VLAN tagging support.

For larger machines, where many domains may be consolidated1, it is possible that most of the communication occurs between domains on the same machine. In this scenario simply forwarding packets back and forth via the service domain is remarkably inefficient. The switch must inspect each packet it receives to determine its destination. Without hardware acceleration this uses CPU resources in the service domain for packets remaining within the physical box.

To improve on latency and lower this overhead expenditure, the virtual networking protocol supports the creation of a distributed switching capability. Where possible, LDCs are created to fully connect all domains that are associated with the same switch. In this way most of the switching burden is moved to the client domain sending the packet. If the destination MAC address of a packet is to a domain with which the guest has a direct LDC link the packet is sent over that link rather than via the switch.

This fully connected distributed switching capability is only possible because the domains are running on a shared memory platform where LDCs are essentially a software creation rather than a scarce hardware resource.

With this capability, the service domain hosting a virtual network switch typically only requires resources to handle broadcast, multicast or upstream packets.

1.7.5. Virtual I/O Limits

There are no architectural limits on the number of services a domain can provide, or on the number of clients that can be serviced. In reality, system resources typically limit practical implementation. Service domains should be sized to accommodate required load and responsiveness. As the virtualized IO is operating by proxy sufficient CPU and memory resources will be required by a service domain to accommodate the load generated by its clients.

This provisioning does not have to be static; if the guest OS in a service domain supports dynamic reconfiguration (“DR”) (see later) then resources can be dynamically added or removed in response to changing loads. It is recommended that operating systems supporting DR are utilized for service domains to provide flexibility in resource assignment and to avoid having to over-provision service domains to accommodate worst-case scenarios.

1.8. Hybrid I/O

For certain I/O devices, for example the in-built network interface unit (NIU) of the UltraSPARC-T2, the underlying hardware consists of multiple functions. In the case of NIU these functions are DMA engines for networking traffic. The intention behind multiple functions is to enable spreading the packet processing load between multiple CPUs— this is important for a unit providing two 10GB/s Ethernet ports.

The direct I/O model allows control of the NIU from a single logical domain where it can be exported as a virtual network interface to other client domains using the virtual I/O model described earlier.

To improve performance the hypervisor also supports a coupled capability, where the device is managed like the virtual I/O model, but with some registers of each device function exported to client domains for higher performance direct I/O access. The combination of a Virtual I/O model with a Direct I/O model for a device is called Hybrid I/O.

The Hybrid I/O model uses the virtual I/O model for device management; in particular the receipt and handling of errors in common device infrastructure. However in addition, direct I/O access is allowed by each client domain to its own subset of the physical device registers. This provides for a higher performance I/O capability without having to use a proxy service.

Physical I/O functions are typically limited to fewer than the possible number of client domains. Therefore, the Hybrid infrastructure is designed to allow a dynamically configurable fall-back to a purely virtual I/O model when hardware functions are needed by other domains. In this way the service domain in a Hybrid I/O model acts as a scheduler for I/O resources switching its client domains between the Virtual and Hybrid modes of device interaction depending on service needs and resource availability. For example, initially 8 DMA engines may be split evenly between 7 client domains and the service domain itself. If another client domain suddenly experiences a high traffic load, all the 7 client DMA engines may be withdrawn and re-assigned to the high load domain.

1.9. Logical Domain Manager

As discussed earlier the hypervisor was architected to be as simple as possible, it provides the machine specific core virtualization functionality and acts as the strict security enforcer between domains.

Management of logical domains requires a set of administrative interfaces (both user and machine) as well as code to ensure correct reconfiguration of the system when domains are created, changed or removed.

To avoid complexity in the hypervisor, this administrative functionality was consigned to an application called the Logical Domain Manager capable of being run on any POSIX compliant guest OS.

The Logical Domain Manager controls the hypervisor and all of the supported logical domains. It provides control interfaces for CLI or automated management interfaces. And, most importantly, it is responsible for the assignment of physical resources to logical domains.

The Logical Domain (LDom) Manager communicates with the hypervisor via a special LDC endpoint called the hypervisor control (“hvctl”) channel. The LDC endpoint is exposed to the user-level Domain Manager via a kernel driver. This LDC endpoint is only accessible from a domain that has been assigned the privilege to control the hypervisor. This is designated the Control Domain.

Other than the Control Domain, no other domain has access to a hypervisor control interface. Since there is no access to a Logical Domain Manager either, other domains in the system are not able to reconfigure the virtual environment or potentially disrupt the machine.

This ensures the strictest possible security for the virtualization control point. Security weaknesses can only be introduced by the system administrator by poor configuration choices. These issues are no different than with any conventional non-virtualized network(s) of machines, and should be familiar to experienced administrators.

1.9.1. Domain roles

From the hypervisor's perspective all domains are the same. We introduce terms describing roles and capabilities only to aid descriptive text. Each and every domain can have one or more of the following roles;

1.9.1.1. I/O domain

An I/O domain has been granted direct access to one or more I/O devices. Typically this provides a limitation on this domain that curtails live-migration to another box unless the guest OS software also supports the ability to dynamically remove the I/O device(s) in support of migration. The LDom Manager should automatically detect and sequence this as required.

1.9.1.2. Service domain

A service domain provides a virtualized (virtual or hybrid I/O) service to other domains in the system. This may be for disk storage or networking, or other future services. A service domain can also be the

client of another service domain. Indeed two service domains can even be each other's client and service respectively. The designation of service domain merely indicates that there is a dependency relationship on this domain by another client domain.

A service domain is often also an IO domain— to provide access to external I/O resources— this is not a requirement. A domain can provide services purely to other clients within the system. For example, a service domain can provide a virtual network (switch) for a number of client domains entirely within the box. For this, no external network interfaces are required, and no need to assign an IO capability.

A service domain can also be the client of another domain's service. For example, a firewall domain may provide a virtual network switch to it clients and at the same time employ a virtual network interface as its upstream link.

1.9.1.3. Control domain

In concrete terms a Control Domain has been granted access to the hypervisor's control interface. If capable of running the Logical Domain Manager it may control and reconfigure the hypervisor and effectively the entire system.

1.9.2. Domain dependencies

A system administrator can define domains to be as dependent or independent of each other as desired within the constraints of available hardware resources.

For example, a very simple configuration of two domains each with direct IO access to its own devices effectively behaves as two entirely separate machines. Essentially these domains can be considered as sharing only the system chassis, power supply and are susceptible only to the most catastrophic of system errors.

At the other end of the spectrum, it is possible to configure all guest domains to have a dependency on a single domain that functions as a combined IO, service and control domain. Even in this extreme single point of failure scenario, this single domain should not crash any of the other client domains if it fails,. Moreover, if it can be rebooted and brought back online the dependent clients should be able to recover.

Key to this is the lack of dependency other domains have on the control domain. The control domain is not required for other guest domains to (re)boot, and can itself be rebooted without affecting any other domains in the system. However, if configuration changes to the hypervisor are required this must be done using the control domain.

From the hypervisor's perspective, there is no special quality or differentiated functionality a role imbibes a domain. Thus, any and all domains can be shutdown, reconfigured or restarted at any time.

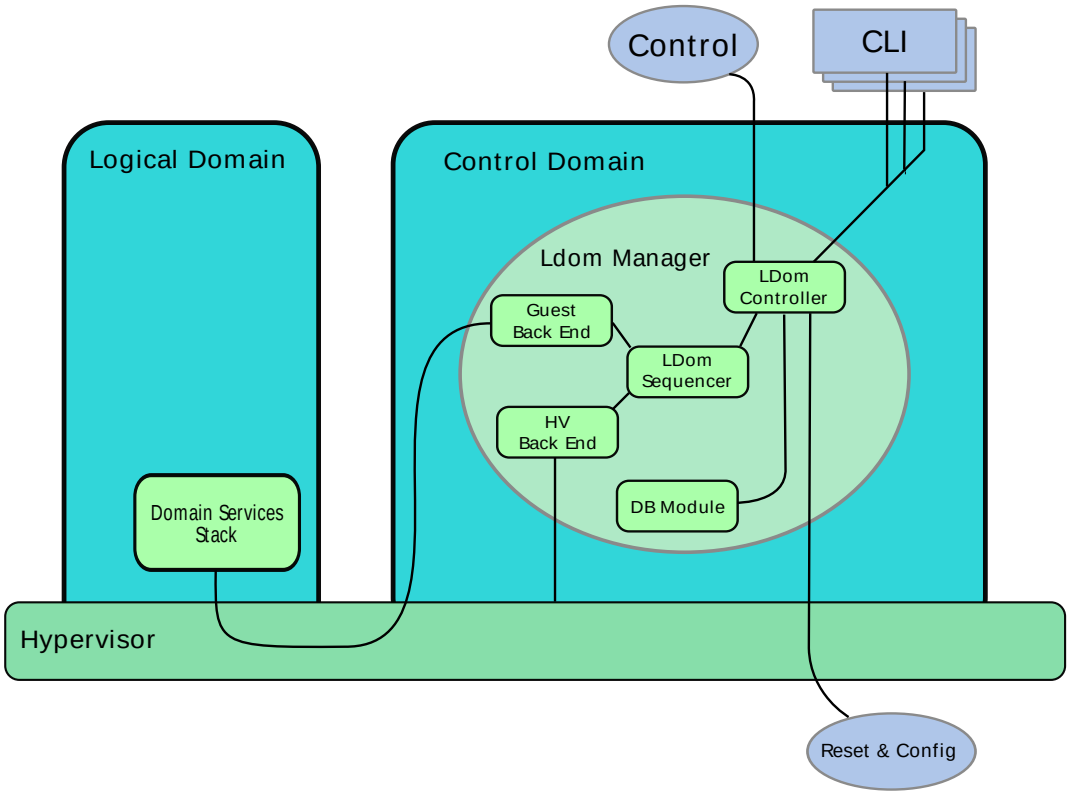
The only system dependencies that exist are created by the system administrator in the configuration of client and service domains. As described earlier the failure of a service domain will exhibit only a temporary outage to a client domain if that service domain can be brought back online. In other words the client does not have to be rebooted with the server. And, if resilience against even temporary outages is sought, multipathed configurations can be created with a single client utilizing two or more service domains.

1.9.3. Domain manager operation

A full description of the internal workings of the Logical Domain Manager is worthy of its own document, and certainly beyond the scope of this one. However it is useful to briefly discuss how the domain manager functions, and how it interacts with each of the logical domains it manages.

A high-level view of the LDom manager is illustrated below.

Figure 1.5. Domain Manager



The LDom manager is a user level application responsible for coordinating and allocating the physical resources of its platform and reconfiguring the hypervisor's internal security rules.

There are two core components to the LDom manager: the LDom controller responsible for domain management and resource assignment decisions, and the the LDom sequencer responsible for sequencing the steps necessary to effect any changes to the overall system.

Typically a system administrator will use a command line or higher level control application to instruct the LDom manager to make configuration changes or to query the state of the virtual machine environment. This is done using a simple TCP/IP socket connection with the control protocol being encapsulated within an XML schema.

The LDom controller receives instructions from its control interfaces and acts upon them using an internal database of resources and domain configuration requirements. The database module itself contains the complete physical resource inventory (“PRI”) of the machine. This inventory is determined by the reset and configuration code run while the system is powering up and retrieved either from the external system controller (or possibly the hypervisor) over a LDC. The PRI itself is in the same binary form as a guest machine description. Although it contains only physical resource information it forms the basis template used to construct the virtual machine descriptions provided to each of the guest domains.

1.9.3.1. Constraint engine

At the heart of the the LDom controller is a constraint engine that assigns resources to domains based upon configuration requirements provided by the administrator. Typically constraints are provided at a high level, such as “5 CPUs” and “1GB of memory”, leaving the constraint engine to pick the most suitable resources using default heuristics appropriate to the platform.

For example, cache sharing information from the PRI can serve to guide the constraint manager in selecting available CPUs that share the same cache for a domain. At the same time it will try to select CPUs that do not share caches with other domains so as to minimize cache interference effects between domains.

If allowed to, by configured constraint rules, the constraint engine may also reconfigure other existing domains to better balance resources in the system.

New configurations are described by new machine descriptions generated by the LDom manager. Each affected domain receives an updated machine description, and the hypervisor is given an update to its hypervisor machine description.

1.9.3.2. Transactional updates

All machine descriptions are downloaded to the hypervisor from the domain manager in a transactional fashion, ensuring that the end state of any reconfiguration operation is either the complete resultant state, or the previous stable state in the case of a configuration error.

By enforcing this transactional model with the LDom manager, the hypervisor can protect itself from unstable or incomplete reconfiguration operations.

Moreover, should the Control Domain fail (crash) part way through a reconfiguration operation, the hypervisor will be left in the previously defined stable state— as if the failed reconfiguration operation had never been attempted.

In the event that the Control Domain or LDom manager do fail, once restarted the LDom manager can retrieve the complete current running state of the virtual machines and available resources from the hypervisor. This key feature enables the control domain to be rebooted arbitrarily without killing or affecting any of the other running domains in the system.

1.9.3.3. Sequencer

After resources are allocated, the LDom controller binds them to the configured domains. Once this step is completed the domain manager proceeds to notify the hypervisor and (appropriate) domains of the change in configuration to the system.

These notifications are delivered via back-end drivers that communicate via LDC to the hypervisor and to live guest domains. Care has to be taken to notify the different parties in the correct order and to ensure correct completion of the transactional model described above.

To achieve this a sequencer in the LDom manager controls the update steps taken during the reconfiguration operation.

For example, adding a CPU to a domain requires first notifying the hypervisor to make the new CPU resource available. Upon completion the domain manager notifies the guest OS in that domain of the availability of the new resource.

A more complex example is the creation of a domain; again the hypervisor is notified first to ensure that the domain is created and resources correctly assigned. Then, any service or other client domains have to be notified to ensure they are aware of the new domain's existence.

Removing resources typically occurs in reverse order, first notifying domains that resources are going away, and when safe notifying the hypervisor to complete the resource reconfiguration.

1.10. Domain service infrastructure

Aside from fundamental services like virtual IO devices, LDCs are also used to connect domains to the domain manager and to other system services.

These channels operate using a “domain services” protocol described later in this document. This protocol enables a domain to advertise its capabilities to the domain manager and to provide non virtual IO services to other domains.

For example, most operating systems cannot easily recover from the unexpected loss of a CPU. So if an operating system is capable of supporting dynamic reconfiguration of CPUs it can announce this capability to the domain manager using the domain services protocol. This serves two purposes; firstly to notify the domain manager that dynamic CPU reconfigurations can be undertaken on this domain while it is running, and secondly to provide a request protocol from the domain manager to the guest to cleanly stop using a CPU resource prior to its removal.

Domain services are negotiated using a common versioned registration protocol, allowing domains to dynamically advertise any reconfiguration operations they are capable of supporting. If a service is not advertised by a domain, the LDom manager infers that it is not safe to undertake the corresponding reconfiguration operation while the guest is running.

Similarly, domain services provide additional proxy capabilities to the domain manager. Thus the domain manager can remotely query domain performance statistics, request reboots or shutdowns. Also, the domain can request changes to its environment variables.

1.11. OpenBoot firmware

Unless otherwise configured for a domain, a virtualized OpenBoot firmware image is provided to each logical domain as it starts. This enables initial loading and execution of an operating system, diagnostic programs, and the ability to configure boot time parameters.

The retention of the OpenBoot command line interface is to maintain compatibility with existing non-virtualized systems. However, for most administrators boot parameter configuration is more easily done when configuring a domain using the LDom manager rather than starting and then logging into the domain's console.

1.12. Error Handling

Handling errors in a virtualized environment poses a number of interesting problems.

Typically errors are delivered via platform specific hardware registers, and correspond to specific hardware resources.

Only the hypervisor can capture these errors, decipher them, and direct them to the affected domain (or worse-case domains) for further recovery.

Ignoring errors often leads to deeper problems such as data corruption. Simply crashing or panicking is not acceptable for a hypervisor that supports multiple virtual machine environments.

The UltraSPARC Hypervisor typically performs the first stage of triage on received errors, collecting the error information (recording for later analysis on the system controller) and then converting this into a virtualized form for delivery to affected domains.

Errors corrected by hardware are typically not reported to affected guest domains. Instead they are recorded for chronic analysis on the system controller. Abnormal correctable error rates can result in the domain manager taking corrective action to avoid using a system resource. For example, CPUs or memory can be pro-actively offlined before they fail.

Uncorrectable errors typically result in some form of data damage. This may be in critical data, (e.g. a kernel data structure), or unused data (e.g. a free page pool) for a guest operating system. The hypervisor

does not know which errors are critical and which are irrelevant, so it reports all uncorrectable errors to the affected domains in a virtualized form.

Use of virtualized error reporting serves two purposes;

Firstly, a guest OS only knows about its virtual resources, not the underlying physical ones. So when reporting a memory error, the hypervisor simply identifies the region of the guest n domain's address space that has become corrupted.

Secondly a guest OS may very well be older than the hardware it is running on. Supplying hardware specific data such as ECC syndromes to a guest operating system is pointless as that OS most likely will not know what to do with the information. Consider a message of the form: “warning temperature 72 degrees”. Without knowledge of the physical hardware, is this a warning of something being too hot or too cold? To avoid these problems error messages have a more precisely defined semantic meaning. For example; “warning: too hot”, or “data corrupted between address X and address Y”.

The information provided to a guest OS is designed to enable the quarantining of affected resources. For example, the off-lining of a corrupted memory page, or at least the (semi) graceful shutdown of the guest OS itself.

Errors can occur as a direct result of a domain action (e.g. a CPU write to memory), or be detected in the background (e.g. via a memory scrubber).

For this reason the hypervisor further categorizes errors into “resumable” and “non-resumable” forms; meaning “after receipt of this error message you can resume what you were doing”, or “you cannot complete what you were doing respectively”.

1.13. Advanced LDoms features

The architectural design of logical domains technology translates into unique capabilities beyond platform virtualization. Logical domains include advanced features that help enterprises ease software migration, simplify reconfiguration of hardware resources, and improve application isolation.

1.13.1. Dynamic reconfiguration

Spikes in demand and changing business needs cause individual IT services to use varying amounts of compute capacity over time. The Logical Domains Manager enables administrators to optimize use of compute resources by modifying the number and type of virtual resources, including CPU, memory, and I/O devices assigned to a logical domain.

The ability to do this is a function of a guest OS's capabilities. However, the domain services mechanism, described earlier, provides an extensible mechanism for a guest to describe those capabilities to the domain manager.

Where the guest OS itself cannot support a dynamic reconfiguration operation, the LDom manager can still support reconfiguring the domain during a reboot of that guest OS. This does not impact any of the other virtual machines in the system. This technique is called “delayed reconfiguration”, and hypervisor updates to a domain's configuration are delayed until the guest OS in that domain shuts down its virtual machine.

1.13.2. Logical domain migration

As mentioned earlier in this section the virtual machine architecture and interfaces have been designed to allow the complete capture of a virtual machine state. This enables a running guest operating system to be frozen and then saved, to be thawed later, or migrated while still running to a different physical machine.

The emphasis on state-less transactional interfaces enables guest domains to be re-bound to new resources arbitrarily. This mechanism is leveraged when moving or saving logical domains.

It falls to the domain manager(s) to ensure that appropriate resources are available at the destination prior to live-migrating a logical domain, but once that determination has been made the operation can proceed until completion.

Snapshots of running domains can be taken to support rapid roll-back or rapid reboot in scenarios where high-availability is paramount. Even for basic domain deployment, a pre-booted snapshot of a domain can be brought online rapidly without having to wait for a guest OS to boot. Dynamic technologies like DHCP can be leveraged to ensure that unique domain characteristics such as host names or IP addresses are dynamically assigned once a “vanilla” snapshot image is started.

Chapter 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 7-bit 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:

Table 2.1. Hyper-fast trap calling convention

Register	Input	Output
%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-bit values 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 64-bit value in register %o5. The calling convention is as follows:

Table 2.2. Fast trap calling convention

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 A.6, “Error codes”).
- 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 A.6, “Error codes”) in %o0.
- All 64 bits of the argument or return values are significant.

Chapter 3. State Definitions

3.1. Processor states

Each virtual CPU can have one of three different states:

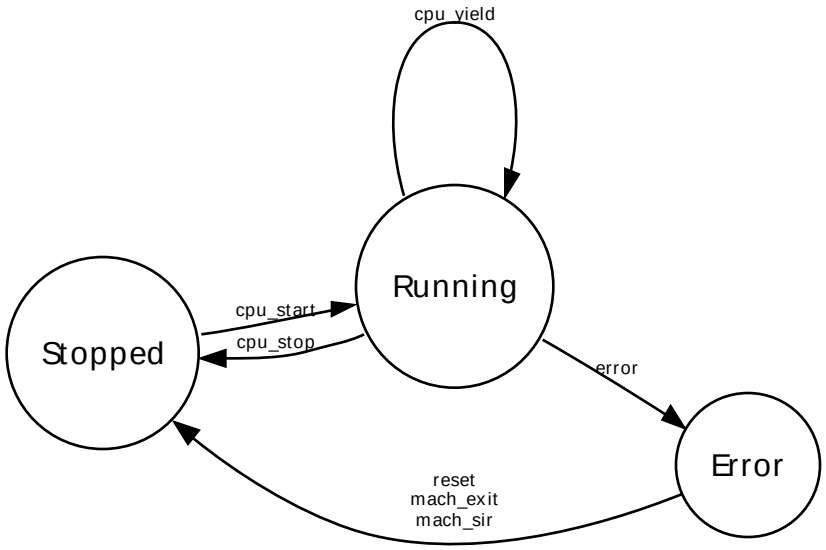
Stopped Stopped CPU is stopped, not executing code, and may be started via the `cpu_start` API service

Running CPU is executing

Error CPU is in error, and is no longer executing code

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

Figure 3.1. Sun4v Processor States



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

Table 3.1. Privileged registers

Register	Initial Value
%cwp	0
%cansave	NWINDOWS - 2
%cleanwin	NWINDOWS - 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 (15)
%tba	current rtba
%tt	POR

3.3.1. Non-Privileged Registers

Table 3.2. Non-privileged registers

Register(s)	Initial Value
%g1-%g7	0
%i0[%cwp]	real address of start-up memory segment
%i1[%cwp]	size of start-up memory segment
%i2-%i7[%cwp]	0
%i0-%i7[all other windows]	0
%l0-%l7[all windows]	0
%f0-%f63	binary 0
%fsr	0

3.3.2. Ancillary State Registers

Table 3.3. Ancillary state registers

Register	Description	Initial Value
ASR 0	%y	0
ASR 2	%ccr	0
ASR 3	%asi	ASI_REAL
ASR 4	%tick	>0, npt=0
ASR 5	%pc	current program counter
ASR 6	%fprs	0

Register	Description	Initial Value
ASR 19	%gsr	0
ASR 22	%softint	0
ASR 24	%stick	>0, npt=0
ASR 25	%stick_cmpr	0 with interrupt disabled (bit 63=1)

3.3.3. Internal memory-mapped registers

Table 3.4. Internal memory-mapped registers

ASI	VA	Initial Value
ASI_SCRATCHPAD	0x00	0
ASI_SCRATCHPAD	0x08	0
ASI_SCRATCHPAD	0x10	0
ASI_SCRATCHPAD	0x18	0
ASI_SCRATCHPAD	0x2, if implemented	0
ASI_SCRATCHPAD	0x28, if implemented	0
ASI_SCRATCHPAD	0x30	0
ASI_SCRATCHPAD	0x38	0
ASI_MMU	0x08 (primary context)	0
ASI_MMU	0x10 (secondary context)	0
ASI_MMU	0x*n*08 (for valid *n*>0)	0
ASI_MMU	0x*n*10 (for valid *n*>0)	0
ASI_QUEUE	0x3c0 (cpu mondo queue head)	0
ASI_QUEUE	0x3c8 (cpu mondo queue tail)	0
ASI_QUEUE	0x3d0 (dev mondo queue head)	0
ASI_QUEUE	0x3d8 (dev mondo queue tail)	0
ASI_QUEUE	0x3e0 (resumable error queue head)	0
ASI_QUEUE	0x3e8 (resumable error queue tail)	0
ASI_QUEUE	0x3f0 (non-resumable error queue head)	0
ASI_QUEUE	0x3f8 (non-resumable error queue 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 `SIS_TRANSITION`, with an empty description string (zeros).

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

Real 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 non-privileged 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:

Table 4.1. Privileged registers

ASI Number	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:

Table 4.2. MMU registers

Register	Address
PRIMARY_CONTEXT*n*	0x*n*08
SECONDARY_CONTEXT*n*	0x*n*10

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.

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.

4.4.4.1. 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`.

4.4.4.2. 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 store to these registers.

Table 4.3. MMU context register barrier rules

	@ 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 effect to be guaranteed observable, otherwise results are undefined
`SECONDARY_CONTEXT`	membar #Sync is required for effect to be guaranteed observable, otherwise results are undefined	membar #Sync, DONE or RETRY are required for effect 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 14.1, “Translation Storage Buffer (TSB) specification ”, or can be established directly by the use of the hypervisor API call `mmu_map_perm_addr` (Section 14.8.7, “mmu_map_perm_addr”). 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_access_mmu_miss`, `data_access_mmu_miss`
: shall be generated when a non-privileged or privileged mode access does not have a translation in any of the TSBs.

`data_access_protection`
: shall be generated when a non-privileged 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_access_exception`, `data_access_exception`
: shall be generated as the result of a non-privileged 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_access_MMU_miss`, `fast_data_access_MMU_miss`
: shall be generated when a non-privileged 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 non-privileged 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 64-byte blocks. The contents and layout of the MMU fault status area are currently specified in Section 14.6, “MMU Fault status area” of this specification.

Chapter 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 privileged-mode trap table

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

Chapter 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 13.2.6, “cpu_qconf”).

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 64-bit quantities. The ASI addresses are as follows:

Table 6.1. Privileged registers

Register	Address	Access
CPU_MONDO_QUEUE_HEAD	0x3c0	Read/Write
CPU_MONDO_QUEUE_TAIL	0x3c8	Read-only
DEV_MONDO_QUEUE_HEAD	0x3d0	Read/Write
DEV_MONDO_QUEUE_TAIL	0x3d8	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.

6.3.1.1. *_QUEUE_HEAD and *_QUEUE_TAIL

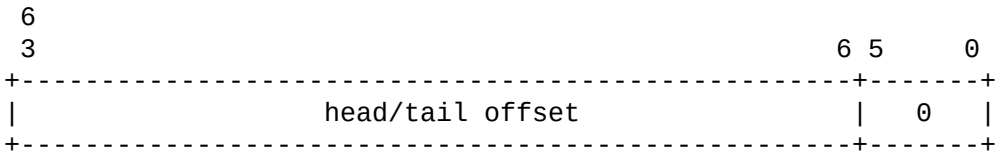
The status of each queue is reflected by its head and tail pointers:

*_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 6.1. Interrupt queue head and tail register formats



The format of each of the `QUEUE_HEAD` and `QUEUE_TAIL` register is shown above. 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.

6.5. Device interrupts

Every device (both virtual and physical) has differing interrupt needs. The device mondo payload was defined to provide a modest amount of information in support of an interrupt so as to minimize the number of additional hypervisor calls required to service an interrupt.

With the device mondo queue registers being implemented by hardware, and directly accessible by the virtual machine's Operating System, no hypervisor API calls are required to identify the source of an interrupt, dispatch the appropriate interrupt handler and subsequently clear the pending interrupt status. Only the device driver itself may need API calls to access the specific device concerned.

6.5.1. Device handles and devinos

To manage devices and their interrupts each device is identified by a device handle. A device handle is unique for a specific device within a virtual machine. The device handle for a device is typically provided to the guest OS running in a virtual machine via the Machine Description (see Chapter 8, *Machine description*) obtainable from the hypervisor. A device handle (or `dev_handle`) should be treated as an opaque cookie value. No semantic information can be derived from the value itself, it is merely a handle by which a guest operating system can identify a device instance to the hypervisor when using an API call.

Devices often have more than one interrupt source. For example, a simple serial device may have separate transmit and receive interrupts. Consequently to identify interrupt sources within a device a second parameter - a device interrupt number or “devino”— is used to disambiguate interrupts belonging to a specific `dev_handle`.

6.6. Sysinos and cookies

As described above, the sun4v virtual machine architecture delivers interrupt notifications to a virtual CPU by means of a device mondo queue. Each interrupt entry in the device mondo queue is a fixed 64 Bytes in size and is used to hold a modest amount of additional information regarding the interrupt it represents.

The first 64-bit word of each device mondo packet holds an identifier for the interrupt source, and the remaining 7 words are defined to be interrupt source specific.

Hypervisor APIs that relate to interrupt handling typically require the passing of a devhandle and the devino to uniquely identify a specific interrupt within the virtual machine.

6.6.1. Legacy use (the sysino)

The initial UltraSPARC T1 hypervisor supplied a “sysino” in word 0 of each device mondo to identify the source of an interrupt. This hypervisor's sysino was derived from the actual device handle and devino of the interrupt source. For the devices in use by a guest operating system the sysinos to be generated by

the hypervisor in device mondos could be determined using the Hypervisor's INTR_DEVINO2SYSINO API call.

The sysino API was intended for the Hypervisor to return a 64-bit value of it's choosing to represent an interrupt source. The arbitrary sysino value was intended such that any algorithm might be employed in generating a sysino for the corresponding device handle and interrupt number. In practice the implementation was simply to concatenate the devhandle and ino values into a single 64-bit sysino number.

Solaris 10 uses this sysino value as an index into a linear table programmed with information relevant to the specific interrupt source. The size of this table fixed at Solaris compile time as a function of the number of CPUs.

The above assumption made by Solaris requires that the sysinos supplied in each device mondo lie in the range 0-2047 - the size of the table when Solaris is compiled for 64 CPUs.

There is no mechanism to enforce this contract between guest OS and hypervisor. The result is simply that the sysinos generated by the hypervisor that are out of range of the table are silently dropped (interrupts are lost), and worse, the upper end of the Solaris table is used for software induced timer interrupts, so unfortunate generation of Hypervisor sysinos can in fact be interpreted as interrupts other than those for the device they represent.

The additional hurdle of dynamic assignment of sysinos presents itself for Logical Domaining and Live Migration. Both features require the ability to dynamically assign and delete interrupt sources for a guest OS, and furthermore transfer those assignments between machines.

Given these and a number of other problems, the sysino interface is being deprecated, and is unlikely to be supported in future hypervisors. New guest operating system code should not use interrupt APIs requiring sysinos unless compatibility with old UltraSPARC-T1 hypervisors is required.

The hypervisor API versioning interfaces can be used to identify the availability of old and new interrupt interfaces when necessary.

As described below the interrupt cookie mechanism that replaces sysinos may be used in a backwards compatible manner to avoid significant re-writes of legacy OS interrupt handling code.

6.6.2. Interrupt cookies

To solve the aforementioned problems with sysinos, Guest OSs and Hypervisor a cookie based mechanism has been implemented.

Instead of a sysino provided by the hypervisor to identify an interrupt source, a guest OS will be able to set a 64-bit cookie value of its choice for a specific devhandle + devino pair. This cookie is returned as word 0 in a device mondo entry when the interrupt occurs. The cookie may be defined and interpreted in anyway by the guest - for example as a pointer to an internal data structure for the interrupt.

Though legacy interrupt sources (for example the existing PCI-E infrastructure on Ontario/Erie) may have cookie support in the Hypervisor, the corresponding guest OS nexus drivers must continue to provide support for existing hypervisor defined sysinos so as to continue to function on legacy firmware implementations.

Similarly, new firmware implementations should continue to provide support for sysino based interrupt APIs, in order to support legacy guest OS nexus drivers.

Chapter 16, *Device interrupt services* of this document defines the APIs used to set and get interrupt cookies in addition to APIs to manipulate the interrupt state machine using by `dev_handle` and ino — thus removing the need for the sysino and the problems of its dynamic allocation and migration between machines.

Chapter 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. 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 write-back error is resumable since the current instruction stream is not affected, but if the corrupted data is later fetched, a non-resumable 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 (non-privileged 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 ASI_QUEUE ASI (0x25). Each of the registers are addressable and accessible as 64-bit quantities. The ASI addresses are as follows:

Table 7.1. Error queue privileged registers

Register	Address	Access
RESUMABLE_ERROR_QUEUE_HEAD	0x3e0	Read/Write
RESUMABLE_ERROR_QUEUE_TAIL	0x3e8	Read-only
NONRESUMABLE_ERROR_QUEUE_HEAD	0x3f0	Read/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. Error Queue Head and Tail Pointers

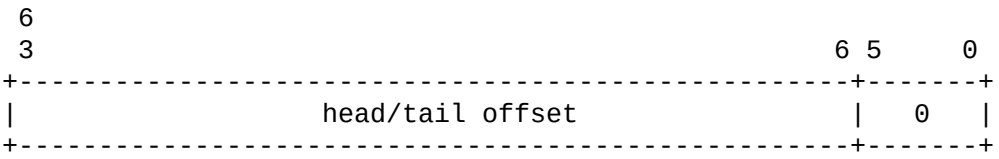
The status of each queue is reflected by its head and tail pointers:

RESUMABLE_ERROR_QUEUE_HEAD, NONRESUMABLE_ERROR_QUEUE_HEAD
: holds the offset to the oldest error report in the queue.

RESUMABLE_ERROR_QUEUE_TAIL, NONRESUMABLE_ERROR_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 7.1. Error queue head and tail register formats



The format of each of the QUEUE_HEAD and QUEUE_TAIL registers is shown above. 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 to 0.

`nonresumable_error`

this trap informs privileged code that an error report has been appended to the non-resumable 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 non-resumable error queue head as quickly as possible to indicate when it is prepared to take another non-resumable error trap. If the non-resumable error queue is not empty when another non-resumable 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.

Chapter 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

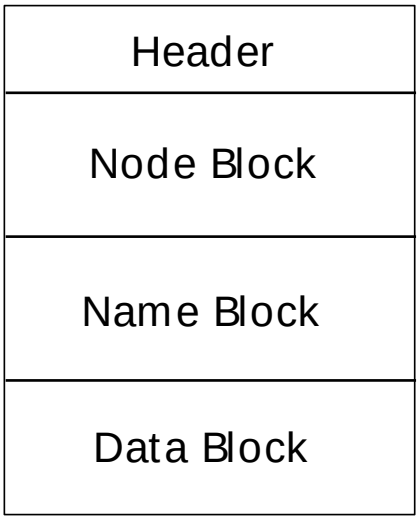
The format of the machine description is designed so that any consumer may either elect to read and transform it into an internal representation, or merely use it in place. For the latter, the encoding needs to be easily readable with an efficient decoder. Similarly a simple encoding requirement also 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 8.1. Machine 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:

Table 8.1. Machine description header

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 32-bit integer value. The upper 16 bits correspond to a major version number, the lower 16 bits 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 232-16 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). `NUL` 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 `NUL` 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 8.13, “Accelerating string lookups”.

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 preceding 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 alignment 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 16 byte 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 16-byte length format comprised of big-endian fields described below:

Table 8.2. Element format

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 NUL character.

The remainder of the node element has two formats depending upon the node's *tag* field. The node element either contains a 64-bit immediate data value, or, for elements requiring an extended data or string, it consists of two 32-bit 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:

Table 8.3. Element tag types

Tag Value	ASCII equiv	Name	Description	Value field
0x00	NUL	LIST_END	End of element list	
0x4e	'N'	NODE	Start of node definition	64-bit index to next node in list of nodes
0x45	'E'	NODE_END	End of node definition	
0x20	SPC	NOOP	No-op list element— to be ignored	0
0x61	'a'	PROP_ARC	Node property arcing to another node	64-bit index of node referenced
0x76	'v'	PROP_VAL	Node property with an integer value	64-bit integer value for property
0x73	's'	PROP_STR	Node property with a string value	offset and length of string within the data block
0x64	'd'	PROP_DATA	Node property with a block of data	offset and length of string within 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 8.19.1, “Root node”) 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. 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.

8.11.1. 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 16 byte 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`.

8.12. 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.

8.12.1. 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 8.19.3, “cpu node” 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 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 */
}
```

8.13. 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 searched 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 64 bits 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 64 bits 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.

8.14. Directed Acyclic Graph

The intrinsic Machine Description (MD) is a collection of directed acyclic graphs (DAGs) 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.

8.14.1. 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 64-bit 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.

Similarly, 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 64-bit 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 located outside a node.

8.15. DAG construction

A DAG is constructed as described above by named 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.

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.

8.16. 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 usable 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.

8.17. 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.

8.18. 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.

8.19. Generic nodes

8.19.1. Root node

Name:	`root`
Required subordinates:	`cpus` (Section 8.19.2, “Cpus node”), `memory` (Section 8.19.4, “Memory node”), `platform` (Section 8.19.6, “Platform node”), `variables` (Section 8.21, “Variables”)
Optional subordinates:	`channel-endpoints` (Section 8.23.6, “Channel endpoints node”), `domain-services` (Section 8.19.7, “Domain services node”), `ioaliases` (Section 8.25.7, “I/O device path aliases collection node”), `keystore` (Section 8.22, “Keystore”), `phys_io` (Section 8.25.1, “Physical Device Collection node”), `virtual-devices` (Section 8.23.2, “Virtual devices node”)

8.19.1.1. 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.

8.19.1.2. Properties

Name	Tag	Required	Description
`content-version`	`PROP_STR`	yes	Version string for the content of this machine description. The currently defined version is “1”.
`md-generation#`	`PROP_VAL`	no	A 64-bit unsigned integer that monotonically increases if the machine description is updated while the domain remains bound, that is, configured within the Hypervisor. A value of zero is to be assumed if this property is absent.

8.19.1.3. Programming note

The purpose of the `md-generation#` number is assist guests that attempt to respond to dynamic updates of their machine descriptions. With the number monotonically increasing a guest is easily able to resolve the temporal ordering of multiple updates of its machine description.

The `md-generation#` values will not to be re-used during the lifetime of the guest domain

8.19.2. Cpus node

Name:	`cpus`
Category:	required by `root`
Required subordinates:	
Optional subordinates:	`cpu` (Section 8.19.3, “cpu node”)

8.19.2.1. 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.

8.19.2.2. Properties

None defined.

8.19.3. cpu node

Name:	`cpu`
Category:	resource required
Required subordinates:	
Optional subordinates:	`exec-unit` (Section 8.20.2, “Exec-unit node”), `cache` (Section 8.20.1, “Cache node”), `tlb` (Section 8.20.3, “TLB node”), `memory-latency-group` (Section 8.24.2, “Memory latency group node”), `pio-latency-group` (Section 8.24.3, “Programmed I/O latency group”), `interrupt-latency-group` (Section 8.24.5, “I/O Interrupt latency group node”)

8.19.3.1. 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`	PRO_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-#ra-bits`	`PROP_VAL`	no	A 64-bit unsigned integer giving the number of real address bits supported by this virtual CPU. If not present, no default value is assumed and the max RA

Name	Tag	Required	Description
			value can be inferred from the `mblock` nodes.
`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.
`mm-#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 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 Section 14.3.2, “TSB entry data word”). 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.
`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.

Name	Tag	Required	Description
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
hwcap-list	PROP_DATA	no	A list of strings identifying which ISA extensions are implemented in this processor. The currently defined values for constructing an hwcap-list are: “ima”, “fjfmau”, “trans”, “random”, “hpc”, “vis3”, “fmau”, “fmaf”, “ASI-BlkInit”, “vis2”, “vis”, “popc”, “v8plus”, “fsmuld”, “div32”, “mul32”.
memory-model-list	PROP_DATA	no	A list of strings identifying which memory models are supported, as per [ua2009] (or future revisions of same) Appendix D (Formal Specification of the Memory Models). Currently defined values are: “tso”, “rmo” and “wc”. These are, respectively, “Total Store Order”, “Relaxed Memory Order”, and “Weak Consistency”.

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 “isalist” are: “sparcv9”, “sparcv8plus”, “sparcv8”, “sparcv8-fsmuld”, “sparcv7”, “sparc”.

Note

Details on the list of currently defined extensions to the SPARC ISA are given in the UltraSPARC Architecture specification [ua2007].

8.19.4. Memory node

Name:	memory
Category:	required by root
Required subordinates:	
Optional subordinates:	mblock (Section 8.19.5, “Mblock node”)

8.19.4.1. Description

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

8.19.4.2. Properties

None defined.

8.19.5. Mblock node

Name:	`mblock`
Category:	required
Required subordinates:	
Optional subordinates:	

8.19.5.1. Description

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

8.19.5.2. Properties

Name	Tag	Required	Description
`base`	`PROP_VAL`	yes	A 64-bit unsigned integer giving the base real address of the memory block represented by this node
`size`	`PROP_VAL`	yes	A 64-bit unsigned integer giving the size in bytes of the memory block represented by this node
`address-congruence-offset`	`PROP_VAL`	no	A 64-bit unsigned integer such that; address-congruence-offset = (PA_base - RA_base) mod M. Where M is a power of 2 strictly greater than all values of `address-mask` and `index-mask` for all the cache and latency group nodes in the MD. See Section 8.24.2.3, “Programming note on RA and physical address congruence”.

8.19.6. Platform node

Name:	`platform`
Category:	core
Required subordinates:	
Optional subordinates:	`latency-groups` (Section 8.24.6, “Latency groups node”)

8.19.6.1. Description

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

8.19.6.2. 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 32 bits must be zero.
mac-address	PROP_VAL	no	A 64-bit unsigned integer in which the lower 48 bits holds the mac address assigned to the virtual machine. The upper 16 bits 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 32 bits 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.
uuid	PROP_STR	no	A string that indicates the UUID of the domain. The format of the string is defined by uuid_unparse(3uuid).
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.
cons-read-buffer-size	PROP_VAL	no	Provides a hint as to the size of the console device's internal input buffering - suitable for the cons_read API call.
cons-write-buffer-size	PROP_VAL	no	Provides a hint as to the size of the console device's internal output buffering - suitable for the cons_write API call.
max-cpus	PROP_VAL	no	The theoretical maximum number of virtual CPUs a guest OS may be

Name	Tag	Required	Description
			assigned. If present, the guest software can assume that it will not see more virtual CPUs than specified by this property. If not present, there is no theoretical limit to the number of virtual CPUs the guest may be assigned. Consequently the guest will have to make a determination for itself as to how many and which of its virtual CPUs it activates. The value is an unsigned 64-bit integer.
`inter-cpu-latency`	`PROP_VAL`	no	This property defines the maximum number of nanoseconds of delay the guest might encounter when two processors attempt to rendezvous (inter-processor communication using interrupts, shared memory, etc.). The value is an unsigned 64-bit integer.
`domaining-enabled`	`PROP_VAL`	no	A 64-bit value indicating the availability of domaining on this platform. Valid values are 0 or 1.

8.19.6.3. Programming notes

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”.

Note: The presence of the `max-cpus` property does not place any requirement on the guest to support the number of virtual CPUs specified. The guest is always free to further constrain the number of virtual CPUs that it will support.

Note: The `inter-cpu-latency` property is intended to bound the amount of time privileged software should consider when calculating timeouts to be used for detecting non-responsive virtual CPUs. This value does not account for additional time required due to the implementation of the privileged code itself, such as executing for prolonged periods with interrupts disabled (pstate.ie==0). The total amount of time imposed by the system added to the amount of time imposed by the guest should be used as the basis for calculating timeout values. More specific latency information may be provided via latency groups in the same machine description see Section 8.24, “Latency nodes”

Note: Platform node properties may be added, removed, or changed at any time, with notification provided by the MD update domain service. Guest software is expected to take notice and accommodate changes when they occur.

Note: The absence of the `domaining-enabled` flag indicates that the platform firmware is not capable of supporting multiple domains. The `domaining-enabled` flag, if present and set to 0, indicates that the platform firmware is capable of multiple domains, however the domain manager has not been used to configure the platform. The `domaining-enabled` flag, if present and set to 1, indicates that the platform firmware is capable of multiple domains and the domain manager may have configured multiple domains on this platform.

8.19.7. Domain services node

Name:	`domain-services`
Category:	optional, under root
Required subordinates:	
Optional subordinates:	`domain-services-port`

8.19.7.1. Description

This construction node leads directly to all the domain services ports supported within this virtual machine. There is only one domain-services node per virtual machine.

8.19.8. Domain services port node

Name:	`domain-services-port`
Category:	optionally required by `domain-services` or `openboot`
Required subordinates:	
Optional subordinates:	`channel-endpoint`

8.19.8.1. Description

This node uniquely represents an instance of a domain services port. The `domain-services` node or `openboot` node will have zero or more `domain-services-port` nodes. A `domain-services-port` under an `openboot` node is intended exclusively for use by OpenBoot firmware.

8.19.8.2. Properties

Name	Tag	Required	Description
`id`	`PROP_VAL`	yes	A 64-bit unsigned integer uniquely identifying this domain service port within the `domain-services` node or `openboot` node.
`md-generation#`	`PROP_VAL`	no	A 64-bit unsigned integer that monotonically increases if the machine description is updated while the domain remains bound, that is, configured within the Hypervisor. A value of zero is to be assumed if this property is absent.

8.20. Memory hierarchy nodes

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

8.20.1. Cache node

Name:	`cache`
Category:	optional
Required subordinates:	
Optional subordinates:	`cache` (Section 8.20.1, “Cache node”)

8.20.1.1. Description

This node describes a cache in the memory system hierarchy.

8.20.1.2. 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”.
index-mask	PROP_VAL	no	A 64-bit unsigned integer. A bit in index-mask is set if that bit in a PA influences the cache index at which a memory is stored when cache resident. This property is discussed later with regard to page coloring in Section 8.24.2.4, “Page coloring”.

8.20.2. Exec-unit node

Name: exec-unit

Category:	optional
Required subordinates:	
Optional subordinates:	`cache` (Section 8.20.1, “Cache node”), `tlb` (Section 8.20.3, “TLB node”)

8.20.2.1. 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.

8.20.2.2. 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 Niagara-2 and Victoria-Falls specific types are: “rng” - Random number generator

8.20.2.3. Programming Note

Some very early releases of Sun firmware included nodes erroneously named “exec_unit” (note underscore instead of dash). Software should ignore these nodes and their contents as in a few cases the information provided was in fact incorrect. Software correctly written to this specification should automatically ignore these false nodes anyway since they are not named “exec-unit”.

8.20.3. TLB node

Name:	`tlb`
Category:	optional
Required subordinates:	
Optional subordinates:	`tlb` (Section 8.20.3, “TLB node”)

8.20.3.1. Description

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

8.20.3.2. 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

Name	Tag	Required	Description
			`"data"` translates data accesses

8.21. Variables

Name: `variables`
Category: required by `root`
Required subordinates:
Optional subordinates:

8.21.1. Description

This machine description node is used to supply variable values to the guest operating system of the virtual machine. These variables are part of the operating environment of the virtual machine and being present in the machine description may be preserved across reboots and power-cycles of the virtual machine and overall system.

Each property in the node constitutes a variable and its value. Variables can be retrieved by name or by retrieving each of the properties of the variables node.

8.21.1.1. Properties

Name	Tag	Required	Description
"variable name"	`PROP_STR`	yes	The variable's value. A `NUL`-terminated string.

8.22. Keystore

Name: `keystore`
Category: optionally required by `root`
Required subordinates:
Optional subordinates:

8.22.1. Description

This node contains a list of security keys used for WAN Boot support. See Section 30.13, “Security key domain service version 1.0”. The node consists of a list of security keys formatted as name and value string pairs. The key names are chosen by the user.

8.22.1.1. Properties

Name	Tag	Required	Description
key name	`PROP_STR`	yes	The security key's value. A `NUL`-terminated string.

The *key name* can be up to 64 characters long and the value for each key can be up to 32 characters long.

The *key name* represents the name of the security key.

8.23. Virtual Devices

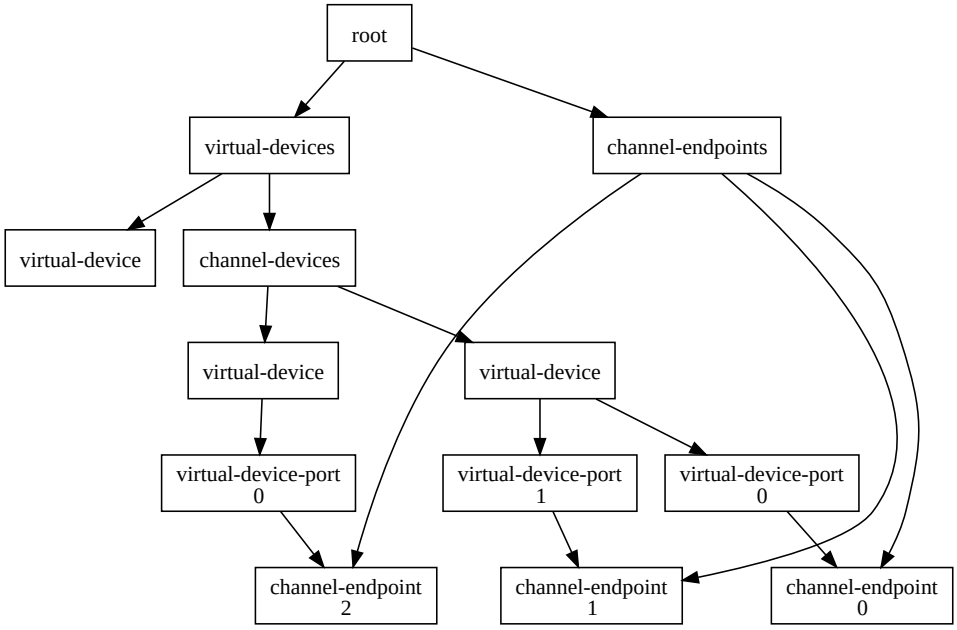
Virtual devices implemented as part of the Virtual IO (VIO) infrastructure are represented in the guest's machine description as nodes together with their properties. This section provides description of these virtual device nodes, the device hierarchy and their properties.

8.23.1. Descriptions for virtual devices

All virtual devices are represented as a node in the guest MD along with its sub-nodes as children of the `virtual-devices` node. All virtual devices nodes are of type virtual-device. The name and compatible properties identify the the specific device and the driver associated with the device. There are two types of virtual device nodes and these are grouped into two separate classes. The first class of device nodes are ones that do not use Logical Domain Channels (LDC) like console, and the existing platform service nodes. These appear as children of the `virtual-devices` node in the MD. All `virtual-device` nodes that use LDCs belong to a class called channel devices and are grouped under a node called `channel-devices`.

>An example node hierarchy for virtual device MD nodes is illustrated above using the “fwd” DAG.

Figure 8.2. Virtual Device hierarchy



The `channel-devices` node is a child of the the `virtual-devices` node. Some of the `virtual-device` nodes under the `channel-devices` node have one or more child port nodes of type `virtual-device-port`. A port for a virtual device represents a communication path to and/or from that virtual device and can be comprised of one or more logical domain channels. Each `virtual-device-port` node can point to one or more `channel-endpoint` nodes corresponding to the logical domain channels within that port.

8.23.2. Virtual devices node

Name:	`virtual-devices`
Category:	optionally required by `root`
Required subordinates:	

Optional subordinates: `virtual-device` (Section 8.23.4, “Virtual device node”) and `channel-devices` (Section 8.23.3, “Channel devices node”)

8.23.2.1. Description

This construction node leads directly to all the virtual devices supported within this virtual machine. The number of instances for each device can be derived by counting the number of nodes for each device.

8.23.2.2. Properties

Name	Tag	Required	Description
`name`	`PROP_STR`	yes	A string name for this node. This value is currently defined as “virtual-devices”.
device_type	`PROP_STR`	`yes`	A string type for this node. This value is currently defined as “virtual-devices”.
`compatible`	`PROP_DATA`	yes	An array of string names for this node. This value is currently defined as “SUNW,sun4v-virtual-devices”.
`cfg-handle`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying this device uniquely.

8.23.3. Channel devices node

Name: `channel-devices`

Category: optionally required by `virtual-devices`

Required subordinates:

Optional subordinates: `virtual-device` (Section 8.23.4, “Virtual device node”)

8.23.3.1. Description

This construction node leads directly to all the channel based virtual devices supported within this virtual machine. The number of instances for each device can be derived by counting the number of nodes for each device.

8.23.3.2. Properties

Name	Tag	Required	Description
`name`	`PROP_STR`	yes	A string name for this node. This value is currently defined as “channel-devices”.
`device-type`	`PROP_STR`	yes	A string type for this node. This value is currently defined as “channel-devices”.
`compatible`	`PROP_DATA`	yes	An array of string names for this node. This value is currently defined as “SUNW,sun4v-channel-devices”.
`cfg-handle`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying this device uniquely.

8.23.4. Virtual device node

Name:	`virtual-device`
Category:	optionally required by `virtual-devices` and `channel-devices`
Required subordinates:	
Optional subordinates:	`virtual-device-port` (Section 8.23.5, “Virtual device port node”)

8.23.4.1. Description

This node uniquely represents an instance of a virtual device. The properties listed here applicable to all virtual devices. Each of the virtual devices may specify additional properties that are device class specific.

8.23.4.2. Common properties

Name	Tag	Required	Description
`name`	`PROP_STR`	yes	Standard property name defining the type of device. See virtual-device class table below.
`type`	`PROP_STR`	yes	Standard property type for this node. See `virtual-device` class table below.
`cfg-handle`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying this device uniquely.
`compatible`	`PROP_DATA`	yes	An array of strings containing compatible device names for this node. See `virtual-device` class table below.

8.23.4.3. Virtual device classes

Table 8.4. Virtual device classes

Service Group	Class	Compatible Name	device-type	name
Console	Client	`SUNW,sun4v-console`	serial	console
Channel Devices				
Network	Client	`SUNW,sun4v-network`	network	network
Network	Server	`SUNW,sun4v-network-switch`	vsw	virtual-network-switch
Block	Client	`SUNW,sun4v-disk`	block	disk
Block	Server	`SUNW,sun4v-disk-server`	vds	virtual-disk-server
Console	Server	`SUNW,sun4v-console-concentrator`	vcc	virtual-console-concentrator
Serial	Server	`SUNW,sun4v-channel`	serial	virtual-channel
Serial	Client	`SUNW,sun4v-channel`	serial	virtual-channel-client
Serial	Server	`SUNW,sun4v-data-plane-channel`	serial	virtual-data-plane-channel

Service Group	Class	Compatible Name	device-type	name
Serial	Client	SUNW,sun4v-data-plane-channel	serial	virtual-data-plane-channel-client
Serial	Server	SUNW,sun4v-domain-service	serial	virtual-domain-service

8.23.4.4. Device class specific properties

Name	Tag	Required	Description
vsw-phys-dev	PROP_DATA	no	An array of string names identifying the physical network devices available locally for use by a virtual switch device
vsw-switch-mode	PROP_DATA	no	An array of string names identifying the order of the preferred switching mode(s) for this switch device. Current valid values are “switched”, “promiscuous”, and “routed”.
local-mac-address	PROP_VAL	no	A 64-bit unsigned integer in which the lower 48 bits hold the mac address assigned to a virtual network or switch device. The upper 16 bits must be zero.
default-vlan-id	PROP_VAL	no	A 64-bit unsigned integer, where the lower 12 bits hold the vlan-id used to designate untagged Ethernet frames set or received by a virtual network or switch device. The upper 52 bits must be zero.
port-vlan-id	PROP_VAL	no	A 64-bit unsigned integer, where the lower 12 bits hold the implicit port vlan-id assigned to this virtual network or switch device. The upper 52 bits must be zero.
vlan-id	PROP_DATA	no	An array of 64-bit unsigned integers, where the lower 12 bits of each element holds the vlan-id(s) assigned to this virtual network or switch device. The upper 52 bits of each element must be zero.
priority-ether-types	PROP_DATA	no	An array of 64-bit unsigned integers, where the lower 16 bits of each element holds a high priority Ethernet type. The upper 48 bits of each element must be zero. The Ethernet type corresponds to the Type field in a Ethernet frame as defined by the Ethernet v2/DIX standard. The virtual network and switch devices should prioritize frames with these types over all other frames, and ensure that these frames are not dropped under congestion.
mtu	PROP_VAL	no	A 64-bit unsigned integer in which the lower 16 bits hold the size of maximum transmission unit (MTU) of a virtual network or a switch device. The upper 48 bits must be zero.

Name	Tag	Required	Description
linkprop	PROP_VAL	no	A 64-bit unsigned integer in which the lower 1 bit holds the information on whether the virtual network or the switch device should attempt to obtain physical link state updates. For a virtual network device, a value of 1 for this bit indicates that it should negotiate for physical link state updates; a value of 0 for this bit indicates that it should not negotiate for physical link state updates. For virtual switch device which is itself configured as an interface, a value of 1 for this bit indicates that it should track physical link state changes and a value of 0 for this bit indicates that it should not track physical link state changes.
vcc-min-tcp-port	PROP_VAL	no	A 64-bit unsigned integer identifying the smallest TCP port assignable to a console group in a SUNW,sun4v-console-concentrator device.
vcc-max-tcp-port	PROP_VAL	no	A 64-bit unsigned integer identifying the largest TCP port assignable to a console group in a SUNW,sun4v-console-concentrator device.
vlds-domain-handle	PROP_VAL	no	A 64-bit unsigned integer that uniquely identifies the domain containing the SUNW,sun4v-domain-service device.
vlds-domain-name	PROP_STR	no	A string that indicates the domain name of the domain containing the SUNW,sun4v-domain-service device.

8.23.5. Virtual device port node

Name:	virtual-device-port
Category:	optionally required by virtual-device node (Section 8.23.4, “Virtual device node”)
Required subordinates:	
Optional subordinates:	channel-endpoint (Section 8.23.7, “Description”)

8.23.5.1. Description

This node uniquely represents an instance of a virtual device port. All virtual-device channels connected to the same client are grouped under a single port device. Every virtual-device has zero or more virtual-device-port nodes.

8.23.5.2. Common properties

Name	Tag	Required	Description
name	PROP_STR	yes	A string name for the device. See virtual-device-port class table.

Name	Tag	Required	Description
id	PROP_VAL	yes	A 64-bit unsigned integer identifying this port uniquely within the virtual-device.

8.23.5.3. Device class-specific port properties

Name	Tag	Required	Description
vds-block-device	PROP_STR	no	A string name identifying the block device used by a port in a SUNW,sun4v-disk-server device.
vds-block-device-opts	PROP_DATA	no	An array of string names identifying the options for the device used by a vds-port in SUNW,sun4v-disk-server device. Current valid options are: "ro" — The device is used and exported by vds as a read-only device. "slice" — The device is exported by vds as a disk slice. "exclusive" — The device is opened for exclusive use by this vds instance only. The device cannot be used by another client or vds instance on the guest. "shared" — The device is exported by the virtual disk server instance to one or more clients connected to it.
vds-block-device-name	PROP_STR	no	A string name identifying the canonical name assigned to the block device used by a port in SUNW,sun4v-disk-server device.
vds-mpgroup-name	PROP_STR	no	A string name identifying the multipath group a port belongs to in a SUNW,sun4v-disk-server device.

Name	Tag	Required	Description
vdc-timeout	PROP_VAL	no	A 64-bit integer identifying a block device's connection timeout. The value specified in seconds determines the period after which a SUNW,sun4v-disk device will timeout submitting requests if it cannot establish a connection with the virtual disk server. If the property is either not specified or set to 0, the block device will wait indefinitely to establish a connection with the virtual disk server.
vcc-tcp-port	PROP_VAL	no	A 64-bit unsigned integer identifying the TCP port assigned to a console group. Provided to the vnts daemon via the vcc driver.
vcc-group-name	PROP_STR	no	A string name identifying the console group for a domain. Provided to the vnts daemon via the vcc driver.
remote-mac-address	PROP_DATA	no	Array of 64-bit unsigned integers where the lower 48-bits of each element holds the mac address assigned to the virtual network or switch device. The upper 16-bits of each element must be zero. This array is a list of mac addresses that are known to be accessible via this port. This is not a complete and comprehensive list.
remote-port-vlan-id	PROP_VAL	no	A 64-bit unsigned integer, where the lower 12-bits holds the implicit port vlan-id assigned to the peer virtual network or switch device. The upper 52-bits must be zero.
remote-vlan-id	PROP_DATA	no	An array of 64-bit unsigned integers, where the lower 12-bits of each element holds the vlan-id(s) assigned to the peer virtual network or switch device. The upper 52-bits of each element must be zero.
maxbw	PROP_VAL	no	A 64-bit unsigned integer identifying the bandwidth limit for this port. The value is specified in bps (bits per second).
switch-port	PROP_VAL	no	Identifies this port as being associated with a SUNW,network-switch device. Property value must be zero. Other values are reserved. Programming note: When using a distributed

Name	Tag	Required	Description
			switch model, this property assists a simple guest in finding a switch port rather than querying every port directly.
`vldc-svc-name`	`PROP_STR`	no	A string name identifying the service a SUNW,sun4v channel device is providing over this port.
`vdpc-svc-name`	`PROP_STR`	no	A string name specifying the service a SUNW,sun4v-data-plane-channel device is providing over this port.
`vlds-remote-domain-handle`	`PROP_VAL`	no	A 64-bit unsigned integer that uniquely identifies the domain to which a `vlds-port` node is associated.
`vlds-remote-domain-name`	`PROP_STR`	no	A string that indicates the domain name of the domain to which a `vlds-port` node is associated.

8.23.5.4. Virtual-device-port class table

Table 8.5. Virtual-device-port classes

Service Group	Class	name	name of parent `virtual-device` node
Network	Client	`vnet-port`	`network`
Network	Server	`vsw-port`	`virtual-network-switch`
Block	Client	`vdc-port`	`disk`
Block	Server	`vds-port`	`virtual-disk-server`
Console	Client	`vcc-port`	`virtual-console-concentrator`
Serial	Server	`vldc-port`	`virtual-channel`
Serial	Client	`vldc-port`	`virtual-channel-client`
Serial	Server	`vdpc-port`	`virtual-data-plane-channel`
Serial	Client	`vdpc-port`	`virtual-data-plane-channel-client`
Serial	Server	`vlds-port`	`virtual-domain-service`

8.23.6. Channel endpoints node

Name: `channel-endpoints`

Category: optionally required by `root`

Required subordinates:

Optional subordinates: `channel-endpoint` (Section 8.23.7, “Description”)

8.23.7. Description

This node uniquely represents a collection of channel endpoint nodes being used by this guest. There should be only one `channel-endpoints` node. The single `channel-endpoints` node will have zero or more `channel-endpoint` nodes as subordinates.

8.23.8. Channel endpoint node

Name:	`channel-endpoint`
Category:	optionally required by `channel-endpoints` node (Section 8.23.6, “Channel endpoints node”) and optionally required by `virtual-device-port` nodes (Section 8.23.5, “Virtual device port node”)
Required subordinates:	
Optional subordinates:	

8.23.8.1. Description

This node uniquely represents an instance of a channel endpoint available to this guest. Every `virtual-device-port` node will have zero or more `channel-endpoint` nodes.

8.23.8.2. Properties

Name	Tag	Required	Description
`id`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying this endpoint uniquely within the virtual machine.
`tx-ino`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying the interrupt number assigned to the transmit interrupt for this endpoint.
`rx-ino`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying the interrupt number assigned to the receive interrupt for this endpoint.

8.23.9. RNG virtual-device node

The RNG hardware support on the UltraSPARC-T2 chip is represented as a single virtual device and is represented in the Machine Description (MD) a `virtual-device` node.

8.23.9.1. Properties

Name	Tag	Required	Description
`name`	`PROP_STR`	yes	"random-number-generator"
`cfg-handle`	`PROP_VAL`	yes	A 64-bit unsigned integer identifying this device uniquely.
`compatible`	`PROP_DATA`	yes	An array of string names for this node. This value is currently de-

Name	Tag	Required	Description
			fined as one of "SUNW,n2-rng", or "SUNW,vf-rng".
rng-#units	PROP_VAL	yes	A 64-bit unsigned integer indicating the number of available RNG devices in the system.

8.23.10. Crypto virtual-device node

The crypto hardware support on the Niagara chip is represented as a single virtual device and is represented in the Machine Description (MD) graph for a Guest as a `virtual-device` node with the following properties:

8.23.10.1. Properties

Name	Tag	Required	Description
name	PROP_STR	yes	The string name for this node is defined as “ncp” or “crypto” for UltraSPARC-T1, as “n2cp” for UltraSPARC-T2.
device-type	PROP_STR	yes	A string type for this node. The value is currently defined as “crypto”, as “n2cp” for UltraSPARC-T2.
device-type	PROP_STR	yes	A string type for this node. The value is currently defined as “crypto”, as “n2cp” for UltraSPARC-T2.
intr	PROP_DATA	yes	List of interrupt numbers. One number per core per type of crypto unit.
ino	PROP_VAL	yes	List of virtual inos generated.
cfg-handle	PROP_VAL	yes	A 64-bit unsigned integer identifying this device uniquely.
compatible	PROP_DATA	no	An array of string names for this node. This value is currently defined as one of "SUNW,sun4v-ncp", or "SUNW,n2-cwq".

8.23.11. MAC-addresses node

Name:	mac-addresses
Category:	optional
Required subordinates:	
Optional subordinates:	mac-address (Section 8.23.12, “MAC-address node”)

8.23.11.1. Description

This node is used to identify fixed mac address resources available to a guest virtual machine. There will be a single `mac-addresses` node that describes all MAC address to device path mappings that a guest OpenBoot can use to allocate MAC address resources. Each forward link of this node will correspond to

a `mac-address` MD node that contains a single device tree pathname and an array of MAC addresses that have been allocated to that device. Each of these `mac-address` nodes may also be a child of any `iodevice`, this allows I/O partitioning by associating an MAC addresses with a particular I/O sub-tree.

8.23.11.2. Properties

This node has no properties but contains forward links to nodes that describe an instance of an MAC address resource in the guest.

8.23.12. MAC-address node

Name:	`mac-address`
Category:	optional
Required subordinates:	
Optional subordinates:	

8.23.12.1. Description

This node contains a device tree path and an array of MAC addresses that have been allocated to that device. See `mac-addresses` node (Section 8.23.11, “MAC-addresses node”)

8.23.12.2. Properties

Name	Tag	Required	Description
`dev`	`PROP_STR`	yes	A string that describes the pathname of a device tree node. This device is being allocated MAC addresses as described by the `mac-addresses` property.
`mac-addresses`	`PROP_DATA`	yes	A consecutive array of six byte elements, each six byte element specifies an 48-bit IEEE 802.3-style MAC address.

8.24. Latency nodes

The following nodes are used to convey latency information to a guest. Latency information may be used by a guest operating system to perform various optimizations within the virtual machine. For example, a guest might optimize the allocation of memory so as to minimize the average access latency for programs running on a particular virtual CPU.

Latency information is provided in the form of latency groups. A latency group node defines the relationship between the MD nodes that lead to it and/or that it leads to.

Four types of latency are defined by this specification:

1. The latency between a virtual CPU and a memory block for load and store operations,

2. The latency between a virtual CPU and a I/O device for load and store operations,

3. The latency between an I/O device and memory for DMA operations, and

4. The latency between an I/O device and a virtual CPU for interrupt delivery.

Physical latency information is provided in each latency group node (defined below) with the latency property. Each latency property value is specified in picoseconds (ps). The actual latency observed in each circumstance may be moderated by the effects of caches and other system components.

Latency group nodes are optional in a machine description. However, for any given type the latency relationships must be full and complete. Thus, if a latency group node describing the load/store latency between one virtual cpu and a memory block exists, then all such latency relationships between all cpus and all memory blocks must be present.

It is recommended, for robustness, that in the event of only partial latency information for a given type being available, a guest should behave as if no latency information of that type is available.

8.24.1. Programming notes and accuracy

Latency information for the types defined above is optional and is not necessarily provided by every virtual platform.

In the event that one of the above types of latency node information is not present in a machine description, a guest operating system must assume a default policy of uniform latency.

A dynamic update to a machine description may add or remove some or all of the latency information. This behavior is to be expected by the guest, which in turn must assume a default uniform latency policy in the event that latency information is not present.

For short transitory periods latency group information presented in a machine description may not reflect the actual relationships of components available to a virtual machine. This can happen, for example, as a result of lag between the reconfiguration of virtual resources and the subsequent machine description update. For this reason, latency group information should only be used for performance optimizations, where inaccuracies may result in sub-optimal performance, but not incorrect behavior.

8.24.2. Memory latency group node

Name:	`memory-latency-group`
Category:	optional
Required subordinates:	`mblock` (Section 8.19.5, “Mblock node”)
Optional subordinates:	

8.24.2.1. Description

This node describes the load and store latency relationship between a virtual CPU and a region of memory. The `memory-latency-group` node is defined to be a optional subordinate of a `cpu` node, and in turn a `mblock` node is defined to be a subordinate of the `memory-latency-group` node.

Thus a search of the “fwd” DAG - starting from a “cpu” node will reveal all the `memory-latency-group` nodes representing that cpu. A search “fwd” from each `memory-latency-group` node will in turn reveal each `mblock` with the described memory latency. So, for example, in the machine description illustrated below we see that CPU 1 can observe `mblock` A with a latency of 100ns, and can observe `mblock` B with a latency of 150ns.

It is common in microprocessor memory system designs to support striped memory addressing, where a number of address bits are used to selected a particular memory bank or chip. Each of these stripes may present a different latency of access for a specific CPU. Often the size of each stripe unit may be quite

small, therefore it is not practicable to provide a mblock for each small stripe so as to connect each to a distinct memory lgroup node.

To resolve the memory striping problem, each memory latency group node holds two additional properties, an address mask (“address-mask”), and an address match (“address-match”) value to be used in conjunction with the real address ranges of the mblocks the latency group nodes connect to.

So, for example, if bit 22 is used to select between two memory banks for a specific cpu - providing a latency strip of 4 M bytes - then two memory-latency-group nodes may connect the `cpu` node to the appropriate mblock node. Both memory-latency-group nodes will have a address-mask property with value 0x400000, with one memory-latency-group node having a address-match property value of 0, and the other memory-latency-group node having a address-match property of 0x400000. Thus the latency information applies to a mblock only for those real addresses where the equation ((address + address-congruence-offset) & address- mask) == address-match holds true. The value address-congruence-offset is a property specified in the mblock corresponding to the specified address, and transforms the address into pseudo address suitable for the mask and match combination.

If address-mask and address-match properties are not present in a memory-latency-group node, then no address striping is in effect, and the described memory latency applies between all mblocks and cpus connected to this memory-latency-group node. The address-mask and address-match properties, while optional, must be provided together. If one property is present without the other a guest must treat the memory-latency- group node as erroneous and ignore it altogether.

8.24.2.2. Properties

Name	Tag	Required	Description
`latency`	`PROP_VAL`	yes	A 64-bit unsigned integer giving the approximate latency of access in picoseconds.
`address-mask`	`PROP_VAL`	no	A 64-bit unsigned integer providing a mask value for a memory stripe.
`address-match`	`PROP_VAL`	no	A 64-bit unsigned integer providing a match value for a memory stripe.

8.24.2.3. Programming note on RA and physical address congruence

The real address space used within a virtual machine is a remapping of portions of a system's underlying physical memory. A guest running within a virtual machine is not provided the physical addresses of its memory blocks. This abstraction of memory addresses enables guests to be moved in memory without changing their real address space layout.

However, to support NUMA and page-coloring algorithms for a guest operating system further information is required that describes the congruency relationship between a real address and the underlying physical address to which it is mapped. To do this, the optional property address-congruence-offset may be optionally added to each mblock node. The property is computed such that:

address-congruence-offset = (PA_base - RA_base) mod M

Where; M is a power of 2 strictly greater than all values of address-mask and index-mask in the MD. A guest operating system must add address-congruence-offset to any real address before applying masks to determine a latency group match, such as address-mask and index-mask.

If this property is not present in the mblock, then its value must be assumed 0. This property is typically provided when the congruency between the real and underlying physical address of a mblock is less than the size needed for lgroup or page color masking. For example; Consider a NUMA machine where memory is striped on 1GB boundaries between 4 different memory controllers. Each cpu may see different access latencies to each of the memory controllers-- each latency is represented by a lgroup node described above. Now consider a 1GB memory segment that starts at real address 0x400000000 and is bound to physical address 0x10000000. To identify 4 different memory controllers with a 1GB stripe the address-mask property of one of the lgroups might have the value 0xc0000000. In this legitimate scenario to correctly apply the lgroup information, the guest OS needs enough correctly congruent bits from the actual physical address to be able to meaningfully apply the lgroup address mask. So for our example, real address 0x400000000 corresponds to physical address 0x10000000, and real address 0x430000000 corresponds to physical address 0x40000000. If we apply the lgroup mask to 0x10000000 we get 0x0. If we apply the lgroup mask to 0x40000000 we get 0x40000000 as the result. Therefore we see that these different address pages reside on different memory controllers with different access latencies. Note: if we had applied the lgroup mask to the corresponding real addresses the result is always 0x0 implying the same memory controller— which would be incorrect.

Thus a means to recover the relevant bits of the physical address are required so that the address mask can be correctly applied. The address-congruence-offset property in an mblock provides this information. As described above the property is derived from the difference between real and their corresponding physical addresses for a mblock. However, to retain ambiguity for actual physical address bindings, this property is not the actual difference, but simply enough bits from the RA/PA difference that an addr mask can be correctly applied. This ambiguity is strictly enforced to prevent guest operating systems being able to bind themselves to specific physical addresses for anti-social activites such as denial of service attacks on specific memory banks or memory controllers on a shared domain platform.

Thus the value provided for address-congruence-offset is sufficient that the equality:

(RA + address-congruence-offset) & address-mask == address-match

holds correctly for all the provided address-mask and address-match values within the MD in order to correctly match lgroups.

If the address-mask 0xc0000000 is the largest mask provided, then the address-congruence-offset for example above would be:

(0x10000000 - 0x400000000) & 0xffffffff = 0x10000000

The address matches for the real addresses above will be,

(0x400000000 + 0x10000000) & 0xc0000000 = 0x0

(0x430000000 + 0x10000000) & 0xc0000000 = 0x40000000

As defined above the address-congruence-offset is an optional property in an mblock node. If not present, a value of 0 can be assumed, thus the equality for matching lgroups reduces to:

RA & address-mask == address-match

8.24.2.4. Page coloring

Page coloring for large caches exhibits a similar set of problems to identifying lgroups.

To assist, a cache node is extended with an optional property index-mask to compute a matching set within the corresponding cache.

The actual cache index employed by hardware is a function of multiple bits from the physical address of the memory reference. To compute a page coloring value the index-mask field identifies the relevant bits from a physical address. Thus the index-bits for page coloring can be derived as:

index-bits = (RA + address-congruence-offset) & index-mask

Where the address-congruence-offset is the property from the mblock (corresponding to the given RA) as described above.

Similarly to lgroup matching, if the address-congruence-offset property is not provided for a mblock its value can be assumed as zero reducing the equation to:

index-bits = RA & index-mask

8.24.3. Programmed I/O latency group

Name:	`pio-latency-group`
Category:	optional
Required subordinates:	
Optional subordinates:	

8.24.3.1. Description

This node describes the access latency of load or store instructions from one or more `cpu` nodes to one or more i/o devices. This node requires at least one subordinate node whose type represents an I/O device, the valid types of these subordinates are listed above in the optional subordinate section.

The pio-latency-group node is defined to be a optional subordinate of a `cpu` node, and in turn each I/O device node is defined to be a subordinate of the `pio-latency-group` node.

The latency information defined by this node may be used to better schedule guest OS functions such as interrupt handlers to virtual cpus with lower latency access to the target devices.

8.24.3.2. Properties

Name	Tag	Required	Description
`latency`	`PROP_VAL`	yes	A 64-bit unsigned integer giving the approximate latency of access in picoseconds.

8.24.4. I/O DMA latency group

Name:	`dma-latency-group`
Category:	optional
Required subordinates:	`mblock` (Section 8.19.5, “Mblock node”)
Optional subordinates:	

8.24.4.1. Description

This node describes the access latency of DMA operations from one or more I/O device nodes to one or more mblocks. This latency information may be used to better allocate memory local to I/O devices where latency of access may be important-- for example in the allocation of device descriptor rings or lookup tables.

The properties describing memory latency and striping are defined as per the memory-latency-group node (see Section 8.24.2, “Memory latency group node”).

8.24.4.2. Properties

Name	Tag	Required	Description
`latency`	`PROP_VAL`	yes	A 64-bit unsigned integer giving the approximate latency of access in picoseconds.
`content-version`	`PROP_STR`	yes	Version string for the content of this machine description. The currently defined version is “1”.
`md-generation#`	`PROP_VAL`	no	A 64-bit unsigned integer that monotonically increases if the machine description is updated while the domain remains bound, that is, configured within the Hypervisor. A value of zero is to be assumed if this property is absent.
`address-mask`	`PROP_VAL`	no	A 64-bit unsigned integer providing a mask value for a memory stripe.
`address-match`	`PROP_VAL`	no	A 64-bit unsigned integer providing a match value for a memory stripe.

8.24.5. I/O Interrupt latency group node

Name:	`interrupt-latency-group`
Category:	optional
Required subordinates:	
Optional subordinates:	

8.24.5.1. Description

This node describes the latency of interrupt delivery from one or more I/O device nodes to one or more cpu nodes. This latency information may be used to better assign virtual cpus to interrupt sources in such cases where low interrupt latency is required. This node is subordinate to `cpu` nodes and to I/O nodes such as `vpci-bus` nodes.

8.24.5.2. Properties

Name	Tag	Required	Description
`latency`	`PRP_VAL`	yes	A 64-bit unsigned integer giving the approximate latency of access in picoseconds.

8.24.6. Latency groups node

Name:	`latency-groups`
Category:	optional
Required subordinates:	
Optional subordinates:	`memory-latency-group` (Section 8.24.2, “Memory latency group node”), `pio-latency-group` (Section 8.24.3, “Programmed I/O latency group”), `dma-latency-group` (Section 8.24.4, “I/O DMA latency group”), `interrupt-latency-group` (Section 8.24.5, “I/O Interrupt latency group node”)

8.24.6.1. Description

This collective node leads to all of the latency group nodes in a guest MD. If any of the memory, PIO, DMA, or IRQ latency group nodes exist in a machine description, then the `latency-groups` node must exist with each of the individual latency group nodes as its subbordinates.

8.24.6.2. Properties

None

8.25. I/O device nodes

These MD nodes describe the static I/O device topology to the OpenBoot guest running in a domain. This allows OpenBoot to extract hardware-specific device information (MAC addresses, interrupt-maps, etc.) from the MD, thereby making the guest hardware agnostic.

The MD iodevice tree is not meant to replace ASR database functionality. If a device is disabled in the ASR database, its node will still appear in the MD, which means that OpenBoot will still have to check if a device is disabled before probing it.

8.25.1. Physical Device Collection node

Name:	`phys_io`
Category:	optionally required by `root`
Required subordinates:	`iodevice` (Section 8.25.2, “I/O device node”)
Optional subordinates:	

8.25.1.1. Description

This node is a collection node referring to the physical devices in the machine description.

8.25.2. I/O device node

Name:	`iodevice`
Category:	optional
Required subordinates:	
Optional subordinates:	`iodevice` (Section 8.25.2, “I/O device node”), `interrupt-map-entry` (Section 8.25.4, “Interrupt mapping node”), `slot-name` (Section 8.25.5, “Slot name node”), `devalias` (Section 8.25.6, “Device

name alias node"), `interrupt-latency-group` (Section 8.24.5, "I/O Interrupt latency group node"), `dma-latency-group` (Section 8.24.4, "I/O DMA latency group") `xaui-mac` (Section 8.25.3, "UltraSPARC-T2 NIU network device node")

8.25.2.1. Description

This node describes properties necessary to create an I/O device node in the OpenBoot device tree.

8.25.2.2. Properties

Name	Tag	Required	Description

`device-type`	`PROP_STR`	yes	A string type for this node. The remaining properties in the MD node for this device will vary depending on the value of this string. For each specific `device-type` allowed, the remaining properties are shown below. The allowed `device-type` values are:
			
			`pciex`
			This value indicates a sun4v root nexus PCI Express device (Fire ports A and B, N2 PIU, etc.)
			
			`pci-switch-upstream`
			This value indicates the node is an upstream port of a PCI Express switch
			
			`pci-switch-downstream`
			This value indicates the node is a downstream port of a PCI Express switch
			
			`pcie-pci-bridge`
			This value indicates the node is a PCI Express to PCI bridge
			
			`pcix-pcix-bridge`
			This value indicates the node is a PCI-X to PCI-X bridge
			
			`pci-network`
			This value indicates the node is a PCI-Express or PCI-X network device
			
			`pci-scsi`
			This value indicates the node is a PCI-Express or PCI-X SCSI adapter

Name	Tag	Required	Description

			`pci-generic`
			This value indicates the node is a PCI-Express or PCI-X device
			
			`sun4v`
			This value indicates the node is a generic sun4v device

8.25.2.2.1. Sun4v to PCI Express root nexus device

The following properties are allowed for device-types that have the value “pciex”.

Name	Tag	Required	Description
`name`	`PROP_STR`	yes	A string with the value “pci”. This value is converted to the OpenBoot device tree `name` property.
`compatible`	`PROP_STR`	yes	A string with the value “SUNW,sun4v-pci”. This value is converted to the OpenBoot device tree `compatible` property.
`cfg-handle`	`PROP_VAL`	yes	A 64-bit value that is unique to this device on the sun4v bus.
`address-ranges`	`PROP_DATA`	*deprecated*, see `ranges`	An array of 64-bit value pairs which specifies the address ranges available to this device. Each pair contains a base and range value. The first pair of the array specifies the PCI IO space addresses, the second pair specifies the PCI 32-bit memory addresses, and the final pair specifies the PCI 64-bit prefetchable memory addresses.
`ranges`	`PROP_DATA`	yes	An array of 7 groups of 64-bit values which specify an entry in the sun4v bus child device's `ranges` property. The first three values represent the `child-phys` address, the second two values represent the `parent-phys` address, and the last two values represent the `size`.
`virtual-dma`	`PROP_DATA`	yes	A pair of 64-bit values which specify the virtual DMA region for this device. This property is converted to the OpenBoot device node's `virtual-dma` property.

Name	Tag	Required	Description
msi-address-ranges	PROP_DATA	yes	An MSI property as specified in [msiprops].
#msi	PROP_VAL	yes	An MSI property as specified in [msiprops].
msi-data-mask	PROP_VAL	yes	An MSI property as specified in [msiprops].
msi-ranges	PROP_DATA	yes	An MSI property as specified in [msiprops].
msi-eq-size	PROP_VAL	yes	An MSI property as specified in [msiprops].
msix-data-width	PROP_VAL	yes	An MSI property as specified in [msiprops].
#msi-eqs	PROP_VAL	yes	An MSI property as specified in [msiprops].
msi-eq-to-devino	PROP_DATA	yes	An MSI property as specified in [msiprops].
bus-ranges	PROP_DATA	yes	A range of PCI bus numbers that this root nexus device can allocate to child PCI devices.
level1-hot-plug-slot-count	PROP_VAL	no	An integer which will be converted into an OpenBoot encoded integer property of the same name.
level2-hot-plug-slot-count	PROP_VAL	no	An integer which will be converted into an OpenBoot encoded integer property of the same name.

8.25.2.2.2. Generic PCI device properties

The following properties are allowed for device-types that have the values: “pcie-switch-upstream”, “pcie-switch-downstream”, “pcie-pcix-bridge”, “pcix-pcix-bridge”, “pci-network”, “pci-scsi”, “pci-generic”.

Name	Tag	Required	Description
device-number	PROP_VAL	yes	The PCI device number for this PCI device
function-number	PROP_VAL	yes	The PCI function number for this PCI device

8.25.2.2.3. PCI bridge device properties

The following properties are allowed for device-types that have the values: “pcie-switch-upstream”, “pcie-switch-downstream”, “pcie-pcix-bridge”, “pcix-pcix-bridge”.

Name	Tag	Required	Description
#interrupt-cells	PROP_VAL	no	This value will be converted into the OpenBoot #interrupt-cells property. The value is the number of 32-bit integers required for the

Name	Tag	Required	Description
			representation of a single interrupt specifier for this node. A more complete description of this property can be found in [ofintrmap].
`interrupt-map-mask`	`PROP_DATA`	no	This array of integers will specify the `interrupt-map-mask` property that OpenBoot needs to create in this PCI device node. A more complete description of this property can be found in [ofintrmap].

8.25.2.2.4. PCI slot device properties

The following properties are allowed for device-types that have the values: “pcie-switch-downstream”, “pcie-pcix-bridge”, “pcix-pcix-bridge”.

Name	Tag	Required	Description
`slot-present`	`PROP_VAL`	no	The presence of this property means that a PCI slot is present at this device/function number.
`hotplug-supported`	`PROP_VAL`	no	The presence of this property means that the slot supports PCI hotplug.

8.25.2.2.5. PCI network device properties

The following properties are allowed for device-types that have the values: “pci-network”.

Name	Tag	Required	Description
`phy-type`	`PROP_STR`	no	This property will contain a string that specifies the type of external physical layer transceiver is connected to the network device. The following are the allowed values: “xgc” for 10Gb copper, “xgf” for 10Gb fiber, “mif” for 1G/100M/10M copper, “pcs” for 1Gb fiber.
`mac-addresses`	`PROP_DATA`	no	An array of MAC addresses allocated to this network device.

8.25.2.2.6. PCI SCSI device properties

The following properties are allowed for device-types that have the values: “pci-scsi”.

Name	Tag	Required	Description
`sas-wwid`	`PROP_VAL`	no	An array of 8 byte SAS WWIDs for this SCSI adapter.

8.25.2.2.7. NIU device properties

The following properties are allowed for UltraSPARC-T2 NIU devices.

Name	Tag	Required	Description
device-type	PROP_STR	yes	A string type for this this node. This value is currently defined as “sun4v”.
compatible	PROP_DATA	yes	An array of string names for this node. This value is currently defined as “SUNW,niumx”.
cfg-handle	PROP_VAL	yes	

8.25.3. UltraSPARC-T2 NIU network device node

Name: xaui-mac

Category: optional, under iodevice

Required subordinates:

Optional subordinates:

8.25.3.1. Description

This node describes the NIU device properties, and the transveiver properties of the external PHY that is connected to the XAUI bus on an NIU port.

8.25.3.2. Properties

Name	Tag	Required	Description
device-type	PROP_STR	yes	A string type for this node. The value is currently defined as “network”.
compatible	PROP_DATA	yes	An array of string names for this node. The value is currently defined as “SUNW,niusl”.
port	PROP_VAL	yes	A 64-bit unsigned integer identifying this device uniquely.
phy-type	PROP_STR	yes	This property contains a string that specifies the type of the external physical layer transceiver that connected to the XAUI bus of this NIU port. The currently defined values are: “xgf” 10Gbps fibre (optical) “xgc” 10Gpbs copper
mac-addresses	PROP_DATA	yes	An array of MAC addresses allocated to this network device.
tx-dma-channels	PROP_DATA	yes	An array of pairs of integers (must contain a multiple of two integers). The first integer of each pair specifies a base transmit DMA channel

Name	Tag	Required	Description
			number, the second integer of the pair specifies then number of transmit DMA channels (beginning at *base*) that the device node has been allocated.
`rx-dma-channels`	`PROP_DATA`	yes	An array of pairs of integers (must contain a multiple of two integers). The first integer of each pair specifies a base receive DMA channel number, the second integer of the pair specifies then number of receive DMA channels (beginning at *base*) that the device node has been allocated.
`interrupts`	`PROP_DATA`	yes	An array of system interrupts allocated for this NIU device.

8.25.4. Interrupt mapping node

Name:	`interrupt-map-entry`
Category:	optional
Required subordinates:	
Optional subordinates:	

8.25.4.1. Description

This node describes a hardware interrupt mapping from a child device interrupt domain to a parent device interrupt domain. I/O device nodes may have forward links to these interrupt mapping nodes. Each node will correspond to a line in the device tree `interrupt-map` property. For more information about the `interrupt-map` related properties, please refer to [ofintrmap].

As stated in the description of `parent-device-path` below, if OpenBoot cannot find the parent interrupt device in the device tree, OpenBoot must eliminate the device tree node corresponding to the `iodevice` node which has the forward arc to the associated `interrupt-map-entry` node. The reason for this is that interrupt mappings cannot span accross multiple domains, so a child interrupt domain must be within the same logical domain as the parent interrupt domain. To avoid forcing restrictions on the device probing order, we will have to overlay interrupt map properties after all devices have been probed.

8.25.4.2. Properties

Name	Tag	Required	Description
`parent-interrupt`	`PROP_DATA`	yes	An array of integers that describes the interrupt number in the parent interrupt domain.
`child-interrupt`	`PROP_DATA`	yes	An array of integers that describes the interrupt number in the child interrupt domain.
`child-unit-address`	`PROP_DATA`	yes	The unit address of the device that generates the interrupt in the child

Name	Tag	Required	Description
			interrupt domain. There number of integers in this array will be equal to the `#address-cells` for the child device.
`parent-device-path`	`PROP_STR`	yes	The textual device path of the device tree node that serves as the parent domain of the interrupt mapping. If the parent device that is specified by the `parent-device-path` property is not present in the device tree, OpenBoot will eliminate the child device node from the device tree.

8.25.5. Slot name node

Name: `slot-name`

Category: optional

Required subordinates:

Optional subordinates:

8.25.5.1. Description

One of possibly multiple slot names for this node. The properties in this node will become an entry in the OpenBoot `slot-names` device tree property. Note that this is an optional node, since not all platforms have implemented the `slot-names` properties for their PCI slots.

8.25.5.2. Properties

Name	Tag	Required	Description
`slot-name`	`PROP_STR`	yes	A `slot-name` entry for the PCI slot MD iodevice node which points to this slot name MD node.
`device#`	`PROP_VAL`	yes	The device number that is associated with the `slot-name` property of this node.

8.25.6. Device name alias node

Name: `devalias`

Category: optional

Required subordinates:

Optional subordinates:

8.25.6.1. Description

One of possibly multiple devaliases associated with this node. The properties in this node will become a *devalias* in OpenBoot.

8.25.6.2. Properties

Name	Tag	Required	Description
devalias	PROP_STR	yes	A device alias for the device path described by the path property of this node.
path	PROP_STR	yes	The device path which will be aliased to the devalias property of this node.

8.25.7. I/O device path aliases collection node

Name: ioaliases
Category: optional
Required subordinates:
Optional subordinates: ioalias (Section 8.25.8, “I/O device path alias node”)

8.25.7.1. Description

This collection node provides forward pointers to all system ioalias nodes

8.25.8. I/O device path alias node

Name: ioalias
Category: optional
Required subordinates:
Optional subordinates:

8.25.8.1. Description

This node provides the current path to a PCIe switch's upstream port and a list of all possible paths to the same PCIe switch upstream port.

8.25.8.2. Properties

Name	Tag	Required	Description
current	PROP_STR	yes	A string containing the path to a PCIe switch upstream port.
aliases	PROP_STR	yes	A list of space separated strings containing all the possible paths to a PCI root nexus.

Chapter 9. Logical domain variables

9.1. Overview

LDom variables control and provide information to the guest's environment. These variables are known as an environmental variables or NVRAM variables on legacy platforms.

These variables are created and consumed by guest software such as OpenBoot. These variables can be modified by guest software CLIs and by LDom manager CLIs. The guest software can create these variables in any data types it chooses. The data types are private to the guest SW itself. Thus, in case of OpenBoot, the formats of the variables are determined by OpenBoot.

The sun4v architecture currently has no predefined variables or values. However OpenBoot software (used by most guest operating environments as their boot loader) does provide a number of environmental variable values.

Rather then push OpenBoot's variable definitions up-stream into the sun4v architecture, OpenBoot (as a layered piece of software) provides default values for these variables itself.

Only when a default value needs to be over-ridden, then the administrator can set a LDom variable of the same name to override the OpenBoot default value.

9.2. LDom variable store

All LDom variables with non-default settings are stored in the LDom variable store and are available to its consumer through machine description (MD). The variable store is managed directly by the LDom manager, and/or indirectly from each guest virtual machine via the variable domain service described in Section 30.12, “Variable Configuration version 1.0”.

If a variable is changed to its non-default value then such change is communicated to LDom manager or to service processor software. The change is reflected in the guest specific machine description (MD). Since only non-default settings are stored in the LDom store, only non-default settings are available in the machine description. All variables not in the machine description are assumed to be set to their default values. The list full list of variables defined by a client and their default values are only known to the client which defines the variables. Typically this client is the OpenBoot firmware.

If the format of the LDom variable in the machine description is not known to its consumer such as OpenBoot then a default value for that variable should be assumed. For example, if OpenBoot does not recognize the value for a variable then the variable will be restored to its default setting.

The non-default settings of all of the LDom variables is communicated using name value string pairs encoded as properties in the variables machine description node.

Even though the values are stored and communicated as name value string pairs, the creators of these variables can create them in any format desired. It is then the responsibility of the consumer of these variables to convert to and from a string encoding for the variable store. For example, if an integer variable is set to `0x0abb0823` then it could be stored in a string format as, “0abb0823”. When the consumer reads the value from the machine description, it should convert the string value back to an integer format. Boolean variables should be converted to either “TRUE” or “FALSE” strings, so that the strings will look exactly the same as user might type at the keyboard. (Though this is convention only and not enforced).

9.3. LDom variables and automatic reboot

Historically there were two ways by which OpenBoot would automatically boot a guest OS. One of the way is by setting the LDom variable `auto-boot?` to TRUE. The second way was valid settings in the

in-memory reboot buffer. For security reasons in LDoms, the concept of the reboot buffer was removed. Three new variables are defined in its place for use with OpenBoot.

OpenBoot's decision to automatic boot a logical domain will be made by first looking at the `reboot-command` variable and then by looking at the `auto-boot?` variable. If the `reboot-command` contains a valid boot string then OpenBoot will execute that boot string command. If the string is null or non-existent then OpenBoot will look for the `auto-boot?` variable. If `auto-boot?` is set to TRUE then OpenBoot will boot the guest OS using the boot device specified by boot-related LDom variables (these are `boot-device` and `boot-file`).

Note: The `diag-device` and `diag-file` variables do not exist on sun4v class platforms.

The following three variables are introduced to support automatic reboot of a guest domain. These variables replace the legacy reboot parameter buffer on non-sun4v platforms.

`reboot-line-number`
: This is an optional variable used by Frame buffer console. The value is a 32-bit integer value describing line number. The default value is 0.

`reboot-column-number`
: This is an optional variable used by Frame buffer console. The value is a 32-bit integer value describing column number. The default value is 0.

`reboot-command`
: This is a required `NUL`-terminated string variable which describes reboot string which includes the boot command, “boot”, optional device path or a device alias and optional file arguments. A null string indicates that the reboot string is not valid. The maximum string length of this variable is 256 characters. The default value of this variable is null. See below for details on the format.

  The string in `reboot-command` is interpreted by OpenBoot as is. The contents of this variable are valid only for one reset. The `reboot-command` string is invalidated by setting it to the null string after OpenBoot has read the variable. If the user wants to set a permanent reboot path and arguments then `auto-boot?` should be set to TRUE together with `boot-path` and `boot-file` being set to the proper device path and boot arguments respectively.

Implementation Note

This variable can be set by using OpenBoot CLIs, guest OS CLIs and also by LDom manager CLIs. It will be updated by the SW responsible for a guest reboot. If OpenBoot is responsible for a guest reboot then it will set the `reboot-command` variable with an appropriate boot string. On legacy platforms, the boot string is stored in a reboot parameters buffer which is part of NVRAM device. If Solaris is responsible for guest reboot then Solaris is responsible for updating this variable directly. In either cases, OpenBoot is the sole consumer of this variable.

9.3.1. Format of `reboot-command` variable

The format of the string in the `reboot-command` consists of the following parameters:

```
boot_command
optional device path or an alias
optional boot arguments
NUL
```

Here, *`boot_command`* is the string “boot”, *`device path`* is the OpenBoot device tree path to the boot device. An *`alias`* is an alias to the boot path. *`Boot arguments`* are arguments passed to the boot command. A `NUL` character terminates the string.

Each of the three parameters above are delineated by one or more space characters (ASCII value `0x20`). If the second parameter is neither a device path (string which starts with "/", ASCII value `0x2f`) or a device alias then the second parameter is the boot argument. The device path can not contain any spaces but boot arguments can have one or more spaces. The end of the boot argument string is the `NUL` character.

Note

If the device path or an alias are not specified then OpenBoot will use the "boot-device" variable value as the boot device. Similarly, if boot arguments are not specified then OpenBoot will use the "boot-file" variable value as boot arguments.

The maximum length of the "reboot-command" variable string is 256 characters. A string consisting of just a `NUL` character (ASCII value `0`) is considered as an invalid boot string.

9.3.2. Guest OS management of LDom variables

A guest OS obtains the list of variables defined by OpenBoot from the `options` device node in the device tree created by OpenBoot. For each such variable, OpenBoot creates properties in the `options` device node. The property contains the name and value for each of the LDom variable. This behavior is the same on all systems that use OpenBoot.

However, guest Operating Systems that retire OpenBoot after booting must manage LDom variables directly if changes are to be stored. Thus the list of LDom variables OpenBoot has defined should be retrieved from the `options` device node. A Guest OS will be able to set any of these variables following the string name/value pair format described above using the variable domain service (Section 30.12, "Variable Configuration version 1.0").

Chapter 10. Security keys

Most sun4v SPARC platforms provide the ability via their OpenBoot firmware boot code to boot a verifiable operating system image across a wide area network (WAN) such as the Internet.

To guard against a “man-in-the-middle” attack where a false boot image is provided in place of a legitimate one for booting, verification and security for boot images is performed using security keys to attest to the correctness of the image being downloaded. OpenBoot documentation provides a more in-depth discussion of this mechanism.

In support of this WAN Boot capability a domain service is provided to be able to store and retrieve these security keys by a LDom on its platform. These keys themselves are typically manipulated via CLIs provided by OpenBoot and operating systems like Solaris.

The WAN Boot key values need to be persisted across reboot. This is achieved in a sun4v virtual machine by presenting the keys in the guest Machine Description (MD) node called “keystore”. Setting and deleting the keys is achieved via a domain service described in this section.

The MD node definitions are given in section Section 8.22, “Keystore”.

The mechanism to store and access the Security Key values is identical to the variable store and access and is described in Section 30.13, “Security key domain service version 1.0”. The only difference is the MD node and the domain services used to access the keys. The keystore format is also identical to LDom variables. The reason for the differentiation is that security keys are not LDom variables and should not be manipulable via the normal variable management CLIs.

Chapter 11. API versioning

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

11.1. API calls

11.1.1. api_set_version

trap#	CORE_TRAP
function#	`API_SET_VERSION`
arg0	*`api_group`*
arg1	*`major_number`*
arg2	*`req_minor_number`*
ret0	*`status`*
ret1	*`actual_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`* 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`* (*`actual_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 returned *`act_minor_number`* is greater than the *`req_minor_number`* then the APIs enabled by the Hypervisor for *`api_group`* will be compatible with *`req_minor_number`*.

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 requested to return to the initial un-set (disabled) state. If the call succeeds it 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 for an *`api_group`*, or requesting that the group return to being unset (*`major_number`*=0), does not reset any previous state associated with services within a group—unless specified explicitly for that group associated state after a `api_set_version` call is undefined.

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`.

6. `api_set_version` does not affect the `CORE_TRAP` API calls - these remain unaffected and may be called at any time.

The API groups are defined in Appendix A, *Number Registry* 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.

11.1.1.1. Errors

EINVAL	The *`api_group`* field is unknown to this hypervisor. This error takes precedence over ENOTSUPPORTED.
ENOTSUPPORTED	If major number for that *`api_group`* is not supported
EOK	If *`api_group`* and *`major_number`* match, or *`major_number`* is zero
EWOULDBLOCK	Operation would block
EBUSY	The *`api_group`* is currently in use, and the requested version would leave the virtual machine in an illegal state

11.1.1.2. 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_set_version` function will return EINVAL. If the major version number for a valid *`api_group`* is not supported the `api_set_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 behavior is to print a warning message and exit the virtual machine.

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

If *Requested* == *Actual*, then the requested minor version is available.

If *Actual* < *Requested*, the guest must be able to determine if the interface with minor version *Actual* offers the required services and proceed accordingly. This is a guest policy issue.

If *Actual* > *Requested*, then the guest may assume it can operate compatibly with version Requested. Minor version number increments are defined to be compatible with the preceding version, so in general the guest may accept *Actual* when *Actual* > *Requested*. 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 *Actual* > *Requested*, the hypervisor may elect to emulate version *Requested*, thus returning *Requested*.

For situations such as the co-residence of OBP with Solaris, or multiple Solaris modules using the same API group, a layered software approach must be taken for version negotiation.

For example, it is recommended that OpenBoot initially negotiate to the lowest version number supported for the firmware consolidation for api groups it intends to use. A subsequent guest operating system may then negotiate versions up for each api group by calling though OpenBoot's CIF interface. Using the CIF interface means OpenBoot will be aware of the version negotiation and can adapt itself accordingly to new api versions, or simply veto requested versions it cannot compatibly upgrade to. If a guest negotiates versions directly with the hypervisor bypassing the CIF, the guest is responsible for retiring OpenBoot and providing OpenBoot services for itself.

11.1.2. 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 set API version for the specified group (see Section 11.1.1, “api_set_version”). In the event that no API version has been successfully set the call returns the error code EINVAL and *`ret1`* and *`ret2`* are set to `0`.

11.1.2.1. Errors

EINVAL	No API version yet successfully set

Chapter 12. Core services

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

12.1. API calls

12.1.1. 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

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.

12.1.1.1. Errors

This service does not return.

12.1.2. 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

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 and use the value returned in *ret1*.

12.1.2.1. Errors

EBADALIGN	Buffer is badly aligned
EINVAL	Buffer length is too small for complete machine description.

ENORADDR	Buffer is to an illegal real address

12.1.3. 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.

12.1.3.1. Errors

This service does not return.

12.1.4. mach_set_watchdog

trap#	FAST_TRAP
function#	MACH_SET_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 re-set the expiration. If the timer expires before being re-set 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.

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

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

The implemented timeout granularity is given by the watchdog-resolution property in the platform node of the guest's machine description (see Section 8.19.6, “Platform node”. The specified timeout is rounded up to the nearest integer multiple of watchdog-resolution milliseconds.

The largest timeout value allowed is specified by the watchdog-max-timeout property of the platform node. 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 milliseconds that were remaining until the timer was to expire. The time remaining will be rounded up to the nearest millisecond of watchdog-resolution available.

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.

12.1.4.1. Errors

EINVAL	*timeout* too large

12.1.5. mach_suspend

trap#	FAST_TRAP
function#	MACH_SUSPEND
ret0	*status*

This call suspends the current virtual machine. The function will return upon the resume. A suspended virtual machine can only be resumed by the domain manager.

All resources that were available to the domain prior to suspension will still be available after resumption, but additional resources may be available. After resumption, a suspended domain's tick/stick may have changed by either a positive or negative offset.

On success, the call returns a status of EOK. Otherwise, it returns one of the following errors.

12.1.5.1. Errors

ENOTSUPPORTED	The requested operation cannot be performed on this domain
EWOULDBLOCK	The requested operation cannot be performed at this time

12.1.6. mach_pri

trap#	FAST_TRAP
function#	MACH_PRI
arg0	*buffer*
arg1	*length*
ret0	*status*
ret1	*length*

This service copies the most current physical resource index (PRI) 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

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 and use the value returned in *ret1*.

12.1.6.1. Errors

EBADALIGN	Buffer is badly aligned
EINVAL	Buffer length is too small for complete machine description.
ENORADDR	Buffer is to an illegal real address
ENOACCESS	Access to the PRI is not permitted
ENOTSUPPORTED	The PRI is not accessible using this API

12.1.7. mach_vars

trap#	FAST_TRAP
function#	MACH_VARS
arg0	*buffer*
arg1	*length*
ret0	*status*
ret1	*length*

This service copies the most current Variable Updates 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 and use the value returned in *ret1*.

Note

The Updates MD delivered to OBP is simply a data structure in the machine description format with a single root node containing string properties named after NVRAM variables and containing their non-default values. The Updates MD is also delivered to the LDom Manager on the mdstore domain service so that it can merge the updates into its own version of the control domain guest MD. When an LDoms configuration is subsequently saved to the SP, the variables in the SP variable store are flushed since they are now represented in the running control domain guest MD and in the stored LDoms configuration.

If the SP is faulted when the host is powered on, when the control domain OBP boots it will fail to register the var-config-backup domain service with VBSC. Any non-default NVRAM variables will be present in the guest MD on this first boot after power-on since hostconfig is able to include the latest non-default variables from the backing store in the guest MD it creates. Moreover, if OBP **setenv** command is used to update a variable, it will fail due to the missing variable domain service, and so there will be no new non-default settings to persist after the SP is faulted.

If any non-default variables are present in the variable backing store but not in the guest MD, they are only retrievable via the Updates MD. This happens if OBP **setenv** is used and then the

control domain is soft reset with **reset-all**. This also happens if Solaris eeprom(1M) is used while the LDom Manager is not running and the control domain is rebooted. There are additional use-cases that can result in this condition. If the SP is faulted while in this condition, absent some recovery, the user will see the OBP variables revert to either a historical value still present in the guest MD or back to the default value.

It is proposed that if the SP is faulted OBP will retrieve the Updates MD via a an hypervisor API call. The Updates MD will be stored in the host flash and kept up-to-date by VBSC. Hypervisor will retrieve the Updates MD from the flash and make it available to the control domain OBP. The layout of the flash and the Updates MD in it is described in the host flash section.

The control domain OBP will continue to use the variable domain service when the SP is present. Non-control domain OBP will continue to only use the variable domain service provided by the LDom Manager. The control domain OBP will only fallback to using the hypervisor API when the var-config-backup domain service fails to register. This might happen for three reasons: 1) the SP is faulted; 2) host is doing parallel boot and VBSC on the SP is not yet running; 3) a hardware or firmware bug causes the ldc/ds transport for the variable service to fail.

While the hypervisor API is necessary for the case where the SP is faulted, it also provides an ideal recovery when the SP is not faulted but still booting. This might happen during a cold power-on boot of host and SP (parallel boot), or it might happen if the host and control domain are rebooted in parallel (parallel reboot).

12.1.7.1. Errors

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

12.1.8. mach_reboot_data_set

trap#	`FAST_TRAP`
function#	`MACH_REBOOT_DATA_SET`
arg0	*`buffer`*
arg1	*`length`*
ret0	*`status`*

This service stores data across a domain reset. The stored data will persist across a MACH_SIR. Initially the stored data will be cleared and the stored data length will be reset to the value zero.

The argument *`length`* is the size of the data in the buffer to be saved. If *`length`* is zero, the argument *`buffer`* is ignored, and any previously saved data is destroyed and the saved data length is set to zero. If the argument *`length`* is non-zero, the argument *`buffer`* is the real address of a data buffer. The buffer provided must be 16-byte aligned.

If this function returns successfully, the saved data and length have been updated. If the argument *`length`* is non-zero, *`length`* bytes of data from the data buffer defined by the argument *`buffer`* have been saved. The saved data and data length may be retrieved via the `mach_reboot_data_get` API.

If this function returns an error, the saved data and data length are unchanged.

The minimum supported size of the internal data buffer shall be 512 bytes and the actual maximum may be higher.

It is the guest's responsibility to clear the reboot data after it has been retrieved. The data may be cleared by calling this API with a *length* of 0.

Note

In SP degraded mode, if a domain is rebooted, Solaris would not be able to save the reboot parameters on the SP. Alternately, if the SP goes down after reboot parameters are saved on the SP but before OBP is able to retrieve the parameters, OBP would not be able to boot Solaris using the specified parameters.

In order to support a guest domain reboot in SP-degraded mode, these APIs are defined. Solaris would use these APIs to save the reboot parameters for the guest. When OBP is ready to boot it would use these APIs to retrieve the reboot parameters and initiate the boot.

Both Solaris and OBP need to negotiate the “Reboot Data Services” API group. If the API group is successfully negotiated they should use only these APIs for reboot command parameters. If the API group is not negotiated they should continue to use the existing method of storing/retrieving the reboot parameters.

12.1.8.1. Errors

EBADALIGN	Buffer is badly aligned
ENORADDR	Buffer is to an illegal real address
EINVAL	The requested *length* is larger than the maximum size

12.1.9. mach_reboot_data_get

trap#	FAST_TRAP
function#	MACH_REBOOT_DATA_GET
arg0	*buffer*
arg1	*length*
ret0	*status*
ret0	*actual-length*

This service returns a copy of the currently saved reboot data. The saved reboot data may be set via the mach_reboot_data_set API.

If the argument *length* is non-zero, the argument *buffer* is the real address of a *length*-sized data buffer. The buffer provided must be 16-byte aligned. This API copies up to *length* bytes of stored data into the data buffer defined by the *buf* argument.

If the argument *length* is zero, the argument *buffer* is ignored.

In all cases, the actual length of the stored data is returned in *actual_length*. If *actual_length* is zero, no data has been copied to the buffer defined by the *buffer* argument, and there is no stored data.

If the argument *length* is non-zero, and less than the size of the stored data, the error code EINVAL is returned and no data is copied.

If the argument *len* is non-zero and greater than or equal to the size of the stored data, the stored data is copied to the buffer defined by the *buffer* argument.

If *length* is larger than *actual_length*, the contents of the buffer beyond *actual_length* is undefined.

Note

The client can easily determine the size of the stored data by calling this service with the *len* argument set to the value 0.

12.1.9.1. Errors

EBADALIGN	Buffer is badly aligned
ENORADDR	Buffer is to an illegal real address
EINVAL	The requested length is larger than the maximum size

Chapter 13. 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.

13.1. CPU id and CPU list

A CPU id is a pre-assigned 16-bit 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.

A valid CPU list must have one or more CPU id entries.

13.2. API calls

13.2.1. 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 *rtba*. The indicated CPU must be in the stopped state. The supplied *rtba* must be aligned on a 256 byte 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.

13.2.1.1. 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

13.2.2. cpu_stop

trap#	FAST_TRAP
function#	CPU_STOP
arg0	*cpuid*

ret0	*status*

Stop CPU *cpuid*. 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.

13.2.2.1. Errors

ENOCPU	Invalid *cpuid*
EINVAL	target *cpuid* is the current CPU
EINVAL	target *cpuid* is not in the running state
EWOULDBLOCK	if starting resource is not available
ENOTSUPPORTED	if not supported on the platform

13.2.3. 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 *rtba*. The supplied *rtba* must be aligned on a 256 byte 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.

13.2.3.1. Errors

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

13.2.4. cpu_get_rtba

trap#	FAST_TRAP
function#	CPU_GET_RTBA
ret0	*status*
ret1	*rtba*

Returns the current value of *rtba* in *ret1*.

13.2.4.1. Errors

No possible error.

13.2.5. 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.

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.

13.2.5.1. Errors

No possible error.

13.2.6. 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_raddr*, and of *nentries* entries. *nentries* must be a power of two. *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 specified queue is unconfigured 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

Queue	Description
0x3e	resumeable 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.

13.2.6.1. Errors

EINVAL	Invalid *queue*
EINVAL	*nentries* not a power of two
EINVAL	*nentries* is less than two or too large
ENORADDR	Invalid *base_raddr* real address
EBADALIGN	*base_raddr* is incorrectly aligned

13.2.7. 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	resumeable 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.

13.2.7.1. Errors

EINVAL	Invalid queue

13.2.8. cpu_mondo_send

trap#	FAST_TRAP
function#	CPU_MONDO_SEND

arg0/arg1	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`.

13.2.8.1. Errors

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

13.2.9. 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.

13.2.9.1. Errors

No errors defined

13.2.10. 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

13.2.10.1. Errors

ENOCPU	Invalid *cpuid*

13.2.11. cpu_tick_npt

trap#	FAST_TRAP
function#	CPU_TICK_NPT
arg0	*enabled*
ret0	*status*

This call enables (*arg0* == 1) or disables (*arg0* == 0) the tick.npt bit on the calling CPU.

On success, the call returns a status of EOK. Otherwise, it returns one of the following errors.

13.2.11.1. Errors

EINVAL	Invalid argument

13.2.12. cpu_stick_npt

trap#	FAST_TRAP
function#	CPU_STICK_NPT
arg0	*enabled*
ret0	*status*

This call enables (*arg0* == 1) or disables (*arg0* == 0) the stick.npt bit on the calling CPU.

On success, the call returns a status of EOK. Otherwise, it returns one of the following errors.

13.2.12.1. Errors

EINVAL	Invalid argument

Chapter 14. 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.

14.1. 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:

Table 14.1. TSB descriptor layout

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 8.19.3, “cpu node”).

14.1.1. 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

Bit	Page Size
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:

Bit	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.

14.1.2. 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 (Section 8.19.3, “cpu node”) of the machine description for each virtual CPU.

14.2. 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.

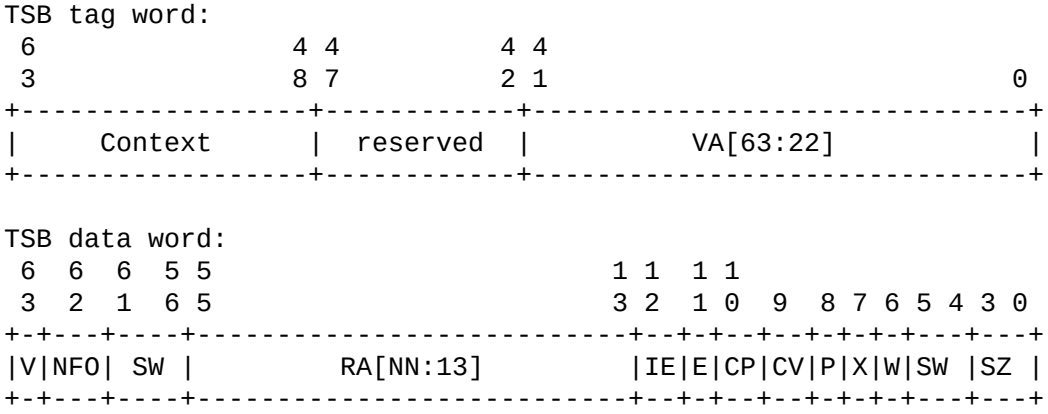
Implementation note

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 demapped, and vice-versa even if not explicitly requested by the flags setting.

14.3. Translation table entries

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

Figure 14.1. Translation Table Entry (TTE) format



Sun4v specifies a TSB entry format with featured as described in the following sections.

14.3.1. TSB entry tag word

The 64-bit TSB entry tag word has a 16-bit 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 42 bits 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 64-bit 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.

14.3.2. TSB entry data word

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

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 bit fields 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 usable 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, non-cacheable 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 non-cacheable 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 to 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.
10-9	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.
6	W	Writable. If the W bit is set to 1, data mapped by this page may be written to.
5-4	SW	Software usable bits

Bit Field	Mnemonic	Meaning
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.

14.4. 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.

14.5. 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.

14.6. 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 64-byte blocks.

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

Table 14.2. MMU Fault Status Area Layout

Offset	Size	Contents
0x00	0x8	Instruction Fault Type (IFT)
0x08	0x8	Instruction Fault Address (IFA)
0x10	0x8	Instruction Fault Context (IFC)

Offset	Size	Contents
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:

Table 14.3. MMU Fault Type values

Code	Fault Type
1	fast MMU miss
2	fast protection fault
3	MMU miss
4	invalid RA
5	privilege 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 (0xffff.ffff.ffff.ffff)	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 a bullet ("•") indicates the field is updated with the relevant fault type, address or context information for the trap).

Table 14.4. MMU Fault Type values

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

sun4v trap type	Fault Type	IFT	IFA	IFC	DFT	DFA	DFC	Comments
								to privileged page (TTE.p=1)
	NFO access (0x7)	•	•	•				instruction access to non-faulty load page (TTE.nfo=1)
	invalid VA (0x9)	•	•	•				instruction virtual access out of range
	invalid TSB entry (0x10)	•	•	•				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)	•	•	•				instruction fetch to real address out of range
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

sun4v trap type	Fault Type	IFT	IFA	IFC	DFT	DFA	DFC	Comments
	invalid page size (0xf)				•			Invalid page size in TTE
	Multiple errors (-1)				•			Hardware encountered multiple errors
data_access_MMU_miss	MMU miss (0x3)				•	•	•	TSB miss
data_access_protection	protection violation (0x6)				•	•	•	store to non-writable mapping
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 protection
privileged_action	privileged action (0xc)					•	•	Use of privileged ASI when pstate.priv=0

14.7. Global MMU Operations

The Global MMU services allow for broadcast demap operations.

When the guest requests one of the global demap services, the API will return a status for that operation. This status will fall into one of the following categories:

Successful operation — The global demap has completed successfully. The status EOK is returned.

Failed operation — The global demap has failed and will never be completed. A status indicating failure is returned. For example, EINVAL, EBUSY, or ENOACCESS has been returned.

Incomplete operation — The global demap operation is still in progress and has not yet either completed successfully or failed permanently. A status of EPENDING is returned.

A guest must not issue a new global demap request while a previous request is in the incomplete state.

A global demap operation is only considered to have completed when a status other than EPENDING is returned. If a global demap operation returns EPENDING, the global demap status service (Section 14.8.18, “mmu_global_demap_status”), must be used to complete the demap operation.

The demap is not globally visible across all processors until the operation has successfully completed.

If the demap completes with any failure status then mappings may still exist in one or more CPUs.

If a status of EBUSY is returned for a demap request, a previous demap operation has not been completed. The guest must use the global demap status service (Section 14.8.18, “mmu_global_demap_status”) to complete the previous demap operation before attempting to initiate a new demap operation.

14.8. API calls

14.8.1. mmu_tsb_ctx0

trap#	FAST_TRAP
function#	MMU_TSB_CTX0
arg0	*ntsbs*
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.

14.8.1.1. Errors

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.
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 page size in a TSB descriptor
EBADTSB	Invalid associativity or size in a TSB descriptor

14.8.2. 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.

14.8.2.1. Errors

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.
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 page size in a TSB descriptor
EBADTSB	Invalid associativity or size in a TSB descriptor

14.8.3. 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 14.2, “MMU flags”.

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

14.8.3.1. 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 *flags* value
ENOTSUPPORTED	*arg0* or *arg1* is non-zero

14.8.4. 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 14.2, “MMU flags”.

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

14.8.4.1. 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 *flags* value
ENOTSUPPORTED	*arg0* or *arg1* is non-zero

14.8.5. 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 14.2, “MMU flags”.

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

14.8.5.1. Errors

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

EINVAL	Invalid *flags* value
ENOTSUPPORTED	*arg0* or *arg1* is non-zero

14.8.6. 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 14.2, “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.

14.8.6.1. 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 *flags* value

EBADPGSZ	Invalid page size value
ENORADDR	Invalid real address in *TTE*

14.8.7. mmu_map_perm_addr

trap#	FAST_TRAP
function#	MMU_MAP_PERM_ADDR
arg0	*vaddr*
arg1	*context*
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 14.2, “MMU flags”.

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

Programming Note: 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.

Programming Note: To effect automatic demap, this service may demap all non-permanent mappings.

14.8.7.1. Errors

EINVAL	Invalid *vaddr* or *flags* value
EBADPGSZ	Invalid page size value
ENORADDR	Invalid real address in *TTE*
ETOOMANY	Too many mapping (maximum of 8 reached)

14.8.8. 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.

The *flags* argument is defined according to Section 14.2, “MMU flags”.

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

14.8.8.1. 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 *flags* value

14.8.9. 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 14.2, “MMU flags”.

14.8.9.1. Errors

EINVAL	Invalid *vaddr* or *flags* value
ENOMAP	Specified mapping was not found

14.8.10. 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 *raddr* 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.

14.8.10.1. Errors

ENORADDR	Invalid real address

EBADALIGN	Invalid alignment of fault area

14.8.11. 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.

14.8.11.1. 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 when already disabled)

14.8.12. 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 *arg0* 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.

14.8.12.1. Errors

ENORADDR	Invalid real address for buffer at *bufferptr*
EBADALIGN	*bufferptr* is badly aligned

EINVAL	supplied buffer size (*maxtsbs*) is too small

14.8.13. 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 *arg0* 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.

14.8.13.1. Errors

ENORADDR	Invalid real address for buffer at *bufferptr*
EBADALIGN	*bufferptr* is badly aligned
EINVAL	supplied buffer size (*maxtsbs*) is too small

14.8.14. mmu_fault_area_info

trap#	FAST_TRAP
function#	MMU_FAULT_AREA_INFO
ret0	*status*
ret1	*fa_ra*

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.

14.8.14.1. Errors

No errors are defined

14.8.15. mmu_global_demap_page

trap#	FAST_TRAP
function#	MMU_GLOBAL_DEMAP_PAGE
arg0	*vaddr*
arg1	*ctx*
arg2	*flags*

ret0	*status*
ret1	*cookie*

This API service demaps any non-permanent page mapping of virtual address *vaddr* in context *ctx* across all CPUs. The *flags* argument is defined according to Section 14.2, “MMU flags”.

14.8.15.1. Errors

EOK	The global demap of the page has completed successfully.
EINVAL	Invalid *vaddr*, *ctx*, or *flags* value.
EBUSY	A new global demap operation cannot be initiated due to previous such operations not having completed.
ENOACCESS	The global demap operations are unavailable.
EPENDING	The newly-initiated global demap operation has been initiated but not completed. The global demap status service (Section 14.8.18, “mmu_global_demap_status”) must be used to determine when the operation has completed.

14.8.16. mmu_global_demap_ctx

trap#	FAST_TRAP
function#	MMU_GLOBAL_DEMAP_CTX
arg0	*ctx*
arg1	*flags*
ret0	*status*
ret1	*cookie*

This API service demaps all non-permanent virtual page mappings with in context *ctx* across all CPUs. The *flags* argument is defined according to Section 14.2, “MMU flags”.

14.8.16.1. Errors

EOK	The global demap of the context has completed successfully.
EINVAL	Invalid *ctx* or *flags* value.
EBUSY	A new global demap operation cannot be initiated due to previous such operations not having completed.
ENOACCESS	The global demap operations are unavailable.
EPENDING	The newly-initiated global demap operation has been initiated but not completed. The global demap status service (Section 14.8.18, “mmu_global_demap_status”) must be used to determine when the operation has completed.

14.8.17. mmu_global_demap_all

trap#	FAST_TRAP
function#	MMU_GLOBAL_DEMAP_ALL
arg0	*flags*

ret0	*status*
ret1	*cookie*

This API service demaps all non-permanent virtual page mappings in all contexts across all CPUs. The *flags* argument is defined according to Section 14.2, “MMU flags”.

14.8.17.1. Errors

EOK	The global demap has completed successfully.
EINVAL	Invalid *flags* value.
EBUSY	A new global demap operation cannot be initiated due to previous such operations not having completed.
ENOACCESS	The global demap operations are unavailable.
EPENDING	The newly-initiated global demap operation has been initiated but not completed. The global demap status service (Section 14.8.18, “mmu_global_demap_status”) must be used to determine when the operation has completed.

14.8.18. mmu_global_demap_status

trap#	FAST_TRAP
function#	MMU_GLOBAL_DEMAP_STATUS
arg0	*cookie*
ret0	*status*

If any of the global demap services return EPENDING, the demap operation has not yet completed across all CPUs. In this case, this service must be invoked using the *cookie* value returned by the previous global demap service. The call must be repeated until it returns success via the *status* value of EOK.

Only when success is returned is it guaranteed that the effects of the global demap are visible across all processors. Failure to wait for completion may result in unpredictable behavior.

This service must be invoked on the same processor that invoked the original global demap service using the *cookie* value returned by that service.

14.8.18.1. Errors

EOK	The global demap corresponding to *cookie* has completed successfully.
EINVAL	Invalid *cookie* value.
ENOACCESS	The global demap operations are unavailable.
EPENDING	The global demap operation corresponding to *cookie* has has not yet completed. Repeat this service.

Chapter 15. 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.

15.1. API calls

15.1.1. 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.

15.1.1.1. Errors

EINVAL	length is zero
ENORADDR	Invalid real address
EBADALIGN	Either the start address or the length are not correctly aligned

15.1.2. 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.

15.1.2.1. Errors

EINVAL	length is zero
ENORADDR	Invalid real address
EBADALIGN	Either the start address or the length are not correctly aligned

Chapter 16. 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.

16.1. 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. Note: this argument is only valid for legacy interrupt interfaces and is considered deprecated. cookie A 64-bit value set by the guest operating system for a specific devhandle, devino combination. Management of cookie values is the responsibility of the guest operating system, and the hypervisor makes no attempt to enforce uniqueness.

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

Table 16.1. Interrupt states

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 interrupt. The state values are defined as:

Table 16.2. Interrupt states

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

16.2. API calls

16.2.1. vintr_getcookie

trap#	FAST_TRAP
function#	VINTR_GETCOOKIE
arg0	*devhandle*
arg1	*devino*
ret0	*status*
ret1	*cookie_value*

This API returns the *cookie_value* that will be delivered in word 0 of a dev_mondo packet to a guest. In the event that no cookie has been set, a value of 0 is returned.

16.2.1.1. Errors

EINVAL	Invalid *devhandle* or devino
ENOTSUPPORTED	(Virtual) device does not support cookies

16.2.2. vintr_setcookie

trap#	FAST_TRAP
function#	VINTR_SETCOOKIE
arg0	*devhandle*
arg1	*devino*
arg2	*cookie_value*
ret0	*status*

Sets the *cookie_value* that will be delivered in word 0 of a dev_mondo packet to a guest. A call to this API will overwrite any previous cookie values set via the same API. If *cookie_value* is 0 the interrupt source is returned to the state of having no cookie assigned, and interrupts are explicitly disabled for the device.

16.2.2.1. Errors

EINVAL	Invalid *devhandle* or *devino*, or *cookie_value* is in range 1..2047
ENOTSUPPORTED	(Virtual) device does not support cookies
EWOULDBLOCK	Operation would block

16.2.3. vintr_getenabled

trap#	FAST_TRAP
function#	VINTR_GETENABLED
arg0	*devhandle*
arg1	*devino*
ret0	*status*

ret1	*intr_enabled*

Returns state in *intr_enabled* for the interrupt specified by *devino*. Return values are: INTR_ENABLED or INTR_DISABLED.

16.2.3.1. Errors

EINVAL	Invalid *devhandle* or *devino*
ENOTSUPPORTED	(Virtual) device does not support the interface

16.2.4. vintr_setenabled

trap#	FAST_TRAP
function#	VINTR_SETENABLED
arg0	devhandle
arg1	*devino*
arg2	*intr_enabled*
ret0	*status*

Sets the *enabled* state of the interrupt *devino*. Legal values for *intr_enabled* are: INTR_ENABLED or INTR_DISABLED.

16.2.4.1. Errors

EINVAL	Invalid *devhandle*, *devino*, or *intr_enabled* value
ENOTSUPPORTED	(Virtual) device does not support the interface

16.2.5. vintr_getstate

trap#	FAST_TRAP
function#	VINTR_GETSTATE
arg0	*devhandle*
arg1	*devino*
ret0	*status*
ret1	*intr_state*

Returns the current state of the interrupt given by the *devino* arguments.

16.2.5.1. Errors

EINVAL	Invalid *devhandle* or *devino*
ENOTSUPPORTED	(Virtual) device does not support the interface

16.2.6. vintr_setstate

trap#	FAST_TRAP
function#	VINTR_SETSTATE
arg0	*devhandle*

arg1	*devino*
arg2	*intr_state*
ret0	*status*

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

Programming note

Setting the state to INTR_IDLE clears any pending interrupt for *devino*.

16.2.6.1. Errors

EINVAL	Invalid *devhandle*, *devino*, or *intr_state* value
ENOTSUPPORTED	(Virtual) device does not support the interface

16.2.7. vintr_gettarget

trap#	FAST_TRAP
function#	VINTR_GETTARGET
arg0	*devhandle*
arg1	*devino*
ret0	*status*
ret1	*cpuid*

Returns the *cpuid* that is the current target of the interrupt given by the *devino* argument. The *cpuid* value returned is undefined if the target has not been set via vintr_settarget.

16.2.7.1. Errors

EINVAL	Invalid *devhandle* or *devino*
ENOTSUPPORTED	(Virtual) device does not support the interface

16.2.8. vintr_settarget

trap#	FAST_TRAP
function#	VINTR_SETTARGET
arg0	*devhandle*
arg1	*devino*
arg2	*cpuid*
ret0	*status*

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

16.2.8.1. Errors

EINVAL	Invalid *devhandle* or *devino*

ENOCPU	Invalid *cpuid*
ENOTSUPPORTED	(Virtual) device does not support the interface

16.3. Deprecated API calls

The following API calls correspond to the legacy *sysino* interrupt interfaces discussed in Section 6.6, “Sysinos and cookies”. These interfaces have now been deprecated. They are documented here (for the time being) for completeness.

16.3.1. 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 interrupt number (*sysino*).

16.3.1.1. Errors

EINVAL	Invalid *devhandle* or *devino*

16.3.2. 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.

16.3.2.1. Errors

EINVAL	Invalid *sysino*

16.3.3. intr_setenabled

trap#	FAST_TRAP
function#	INTR_SETENABLED
arg0	*sysino*
arg2	*intr_enabled*
ret0	*status*

Sets the *enabled* state of the interrupt *sysino* legal values for *intr_enabled* are: INTR_ENABLED or INTR_DISABLED.

16.3.3.1. Errors

EINVAL	Invalid *sysino* or *intr_enabled* value

16.3.4. 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.

16.3.4.1. Errors

EINVAL	Invalid *sysino*

16.3.5. intr_setstate

trap#	FAST_TRAP
function#	INTR_SETSTATE
arg0	*sysino*
arg2	*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*.

Programming note

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

16.3.5.1. Errors

EINVAL	Invalid *sysino* or invalid *intr_state*

16.3.6. 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`.

16.3.6.1. Errors

EINVAL	Invalid *sysino*

16.3.7. 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*.

16.3.7.1. Errors

EINVAL	Invalid *sysino*
ENOCPU	Invalid *cpuid*

16.4. Interrupt API version control

In introducing the interrupt cookie based interrupt API calls, the legacy interrupt interfaces needed to be deprecated. This is achievable using the version negotiation APIs.

However the legacy sysino interfaces were grouped with the core hypervisor APIs (group 0x1).

To resolve this problem, all the interrupt interfaces are now moved to a new group (group 0x2). The legacy (deprecated) API functions will be available to a guest when it negotiates version 1.0 in this group.

The list of APIs being migrated to group 0x2 are as follows:

```
intr_devino2sysino
intr_getenabled
intr_setenabled
intr_getstate
intr_setstate
intr_gettarget
intr_settarget
```

The behavior of these APIs will not change and they will continue to function as described. A guest has to negotiate version 1.0 in group 0x2 prior to accessing these APIs. The new interrupt APIs specified above allow a guest to specify a single 64-bit cookie that will be delivered in the first word (word 0) of a dev_mondo packet. These APIs use the *devhandle* and *devino* to refer to the interrupt source instead of the *sysino* provided by the Hypervisor via the `intr_devino2sysino` API.

The new interrupt API functions will be available to a guest when it negotiates version 2.0 in the interrupt API group 0x2. When a guest negotiates version 2.0, all interrupt sources will only support using the cookie interface, and any attempt to use the version 1.0 interrupt APIs numbered `0xa0` to `0xa6` will result in

the ENOTSUPPORTED error being returned. Interrupts from all sources are explicitly disabled until the guest that negotiated v2.0 in group 0x2, sets a valid cookie value for the interrupt source.

A guest may upgrade to using the cookie based interrupt APIs, by negotiating version 2.0 in group 0x2, even if it had previously negotiated version 1.0 in group 0x2. Subsequent accesses to v1.0 interrupt APIs in group 0x2 will fail with ENOTSUPPORTED. Two different guests running in a system can negotiate different versions in API group 0x2, but a single guest can negotiate either version 1.0 or 2.0 in group 0x2 and use the corresponding APIs.

Chapter 17. 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.

17.1. API calls

17.1.1. 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.

17.1.1.1. Errors

EWOULDBLOCK	TOD resource is temporarily unavailable
ENOTSUPPORTED	TOD resource not supported

17.1.2. tod_set

trap#	`FAST_TRAP`
function#	`TOD_SET`
arg0	*`time-of-day`*
ret0	*`status`*

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

17.1.2.1. Errors

EWOULDBLOCK	TOD resource is temporarily unavailable
ENOTSUPPORTED	TOD resource not supported

Chapter 18. Console services

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

18.1. API calls

18.1.1. 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.

18.1.1.1. Errors

EWOULDBLOCK	No character available

18.1.2. cons_putchar

trap#	FAST_TRAP
function#	CONS_PUTCHAR
arg0	*character*
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.

18.1.2.1. Errors

EINVAL	Illegal character
EWOULDBLOCK	Output buffer currently full, would block

18.1.3. cons_read

trap#	FAST_TRAP
function#	CONS_READ
arg0	*raddr*
arg1	*size*
ret0	*status*
ret1	*retval*

Reads up to *size* characters from the console device and places them in the buffer provided starting at the real address *raddr*.

On success, *status* contains either a special value (as per cons_getchar) or the number of characters placed into the supplied buffer. The number of characters returned may be less than or equal to the buffer size specified. If *ret0* is not EOK, then no characters (special or otherwise) have been read and *retval* is invalid.

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

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

18.1.3.1. Machine description properties

A optional property cons-read-buffer-size in the machine description's platform node provides a hint as to the size of the console's internal input buffering. A guest OS may use this property in determining the appropriate size of the read buffer to pass to this API call.

18.1.3.2. Errors

ENOADDR	Invalid real address
EWOULDBLOCK	Cannot complete operation without blocking
EIO	I/O error

18.1.4. cons_write

trap#	FAST_TRAP
function#	CONS_WRITE
arg0	*raddr*
arg1	*size*
ret0	*status*
ret1	*retval*

Writes up to *size* characters to the console device from the buffer provided starting at the real address *raddr*.

On success, *retval* contains the actual number of characters written to the console device, which may be fewer than the requested number of characters.

If *status* is not EOK, then no characters have been written to the console device and *retval* is invalid.

18.1.4.1. Machine description properties

A optional property cons-write-buffer-size in the machine description's platform node provides a hint as to the size of the console's internal output buffering. A guest OS may use this property in determining the appropriate size of the write buffer to pass to this API call.

18.1.4.2. Errors

ENOADDR	Invalid real address
EWOULDBLOCK	Cannot complete operation without blocking

EIO	I/O error

Chapter 19. Domain state services

This section describes the API services provided for a guest to report its operational state to an external entity.

19.1. API calls

The following API services are provided to get and set the current domain state.

19.1.1. soft_state_set

trap#	FAST_TRAP
function#	SOFT_STATE_SET
arg0	software_state
arg1	software_description_ptr
ret0	status

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 `soft_state_get` 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.

Table 19.1. Guest Software States

Name	Value	Definition
SIS_NORMAL	1	Guest software is operating normally
SIS_TRANSITION	2	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 32-byte 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. This string is to be defined by the guest software— no registry or convention is defined by this API, and guest software is free to use any appropriate string value.

Once the soft-state API group has been successfully negotiated the initial soft state is set to `SIS_TRANSITION` with an empty string for the software description.

19.1.1.1. Errors

EINVAL	*`software_state`* is not valid, or *`software_description`* is not NUL-terminated
ENORADDR	*`software_description`* is not a valid real address
EBADALIGN	*`software_description`* is not correctly aligned

19.1.1.2. 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”

This service is enabled by successfully negotiating a version of its API service group.

Before the group has been enabled a hypervisor may externally report the guest state as unavailable or as `SIS_NORMAL` (with a default string such as “operating normally” depending upon implementation. The current soft state is not visible to the guest itself until the service is enabled.

Once the soft state group has been enabled, the initial state is set to `SIS_TRANSITION` with an empty string. The virtual machine soft 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`.

The state strings used by a guest are to be defined within the context of that guest software, there are no commonly defined strings to be used by all guests. The intended use of the soft state strings is as presentation messages to human readers. Use of commonly defined strings is strongly discouraged so as to prevent interpretation and use by external automated management software. External management software should only ascribe meaning to the well defined software state values.

19.1.2. soft_state_get

trap#	`FAST_TRAP`
function#	`SOFT_STATE_GET`
arg0	software_description_ptr
ret0	status
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 `soft_state_set`).

19.1.2.1. Errors

ENORADDR	*`software_description`* is not a valid real address

EBADALIGN	*`software_description`* is not correctly aligned

Chapter 20. 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 usable by sun4v code. Some platform hypervisors may provide no dump buffer information for security reasons.

20.1. API calls

20.1.1. dump_buf_update

trap#	FAST_TRAP
function#	DUMP_BUF_UPDATE
arg0	raddr
arg1	size
ret0	status
ret1	minsize

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.

20.1.1.1. Errors

EINVAL	*size* is non-zero but less than the minimum size required
ENORADDR	*raddr* is not a valid real address
EBADALIGN	*raddr* is not aligned on a 64-byte boundary
ENOTSUPPORTED	not supported for the current logical domain

20.1.2. dump_buf_info

trap#	FAST_TRAP
function#	DUMP_BUF_INFO
ret0	status
ret1	raddr
ret2	size

This service returns the currently configured dump buffer description.

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

20.1.2.1. Errors

No errors defined.

Chapter 21. 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.

21.1. 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:

Table 21.1. Trap Trace Control Structure

Offset	Size	Contents
0x00	8	Head offset
0x08	8	Tail offset
0x10	48	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.

21.2. Trap trace buffer entry format

Trap trace entries all have the following format:

Table 21.2. Trap Trace Buffer Entry Structure

Offset	Size	Name	Description
0x00	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	Globals level
0x04	2	TTRACE_ENTRY_TT	Trap type
0x06	2	TTRACE_ENTRY_TAG	Extended trap identifier
0x08	8	TTRACE_ENTRY_TSTATE	Privileged 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 state
0x30	8	TTRACE_ENTRY_F4	Entry-specific
0x38	8	TTRACE_ENTRY_F4	Entry-specific

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

Table 21.3. Trap Trace Entry Types

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

21.3. API calls

21.3.1. 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 64-byte aligned.

The *nentries* argument 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.

21.3.1.1. Errors

EINVAL	*nentries* is too small
ENORADDR	*raddr* is not a valid real address
EBADALIGN	*raddr* is not aligned on a 64-byte boundary

21.3.2. ttrace_buf_info

trap#	FAST_TRAP
function#	TTRACE_BUF_INFO
ret0	status

ret1	raddr
ret2	nentries

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) entries.

21.3.2.1. Errors

No errors defined.

21.3.3. ttrace_enable

trap#	FAST_TRAP
function#	TTRACE_ENABLE
arg0	enable
ret0	status
ret1	previous_enable

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.

21.3.3.1. Errors

EINVAL	No buffer currently defined

21.3.4. 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 unfreeze all tracing. The returned *`previous_state`* is 0 for not frozen, and 1 for frozen.

21.3.4.1. Errors

EINVAL	No buffer currently defined

21.3.5. ttrace_addentry

trap#	TTRACE_ADDENTRY
arg0	tag (16-bits)
arg1	data word 0
arg2	data word 1

arg3	data word 2
ret0	status

This function adds an 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.

21.3.5.1. Errors

EINVAL	No buffer currently defined

Chapter 22. 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).

22.1. 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 (64-byte) 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.

22.2. 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.

22.3. 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.

22.4. API calls

The following API calls are provided for LDC usage.

22.4.1. 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 identical value indicating that the queue initially is empty.

The endpoint's transmit queue is unconfigured 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 unconfiguring it. Reconfiguring or unconfiguring 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 reconfiguration 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.

22.4.1.1. Errors

EINVAL	*nentries* not a power of two in number or, *nentries* is less than two or too large.

ENORADDR	*base_raddr* is not a valid real address
EBADALIGN	*base_raddr* is not correctly aligned for its size
ECHANNEL	Invalid *ldc_id*

22.4.2. 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.

22.4.2.1. Errors

ECHANNEL	Invalid *ldc_id*

22.4.3. ldc_tx_get_state

trap#	FAST_TRAP
function#	LDC_TX_GETSTATE
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

22.4.3.1. Errors

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

22.4.4. 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 specified *tail_offset* must be aligned on a 64-byte 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).

22.4.4.1. Errors

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

22.4.5. 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 unconfigured 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 preceding 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 retransmission, or to ensure that no datagrams are pending on the peer endpoint's transmit queue prior to the configuration operation.

22.4.5.1. Errors

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

22.4.6. 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.

22.4.6.1. Errors

ECHANNEL	Invalid *ldc_id*

22.4.7. 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

22.4.7.1. Errors

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

22.4.8. 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 64-byte 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.

22.4.8.1. Errors

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

22.5. Shared Memory API calls

The ldc_set_map_table, ldc_get_map_table, and ldc_copy APIs are usable when version 1.0 of the API group is negotiated.

The ldc_mapin, ldc_unmap and ldc_revoke APIs are available in addition to the 1.0 APIs when version 1.1 of the API group is negotiated.

22.5.1. ldc_set_map_table

trap#	FAST_TRAP
function#	LDC_SET_MAP_TABLE
arg0	channel
arg1	base_ra
arg2	nentries

ret0	status

This API service enables a guest to declare an export map table and bind that map table to the specified logical domain *channel*.

The map table must consist of a power of two number of entries specified by the *nentries* argument. The minimum number of entries is 2. The export map table base real address is specified in *base_ra* and must be aligned to the same boundary as the overall size of the table in bytes (*nentries**8).

Specifying zero (0) for *nentries* unbinds any map table previously bound to the domain channel. If *nentries* is zero, *base_ra* is ignored. Unbinding a map table does not automatically revoke exported pages, any pages still in use by an importing domain may remain accessible by that domain for an indeterminate period of time, or until the exporting domain exits.

22.5.1.1. Errors

EINVAL	*nentries* is invalid, or specified domain *channel* does not support a shared memory interface
ENORADDR	Invalid *base_ra* or real address range for the map table
EBADALIGN	Map table *base_ra* is not correctly aligned for the size of the table
ECHANNEL	Invalid domain *channel*
EWOULDBLOCK	Operation would block

22.5.2. ldc_get_map_table

trap#	FAST_TRAP
function#	LDC_GET_MAP_TABLE
arg0	channel
ret0	status
ret1	base_ra
ret2	nentries

This API service retrieves the current map table configuration associated with the given domain *channel*.

If no map table is configured, both *base_ra* and *nentries* are returned as zero.

22.5.2.1. Errors

ECHANNEL	Invalid domain *channel*
EWOULDBLOCK	Operation would block

22.5.3. ldc_copy

trap#	FAST_TRAP
function#	LDC_COPY
arg0	channel
arg1	flags
arg2	cookie

arg3	raddr
arg4	length
ret0	status
ret1	ret_length

This API service copies data into or out of a local memory region from or to the logical domain at the other end of the specified domain *channel*.

The local memory buffer to be used is a contiguous real address buffer starting at *raddr*, and of size *length*. Both *raddr* and *length* must be aligned to 8-byte boundaries. The exported page to be accessed by the copy operation is identified by *cookie*.

A copy-in or copy-out operation is specified by the *flags* argument, for which the following values apply:

LDC_COPY_IN	0	Copy from remote exporting domain into buffer of local domain
LDC_COPY_OUT	1	Copy from buffer of local domain into page exported by remote domain

All other values for flags are illegal.

In the event of success, the return *status* is EOK, and *ret_length* contains the actual number of bytes copied. In this event 0 <= *ret_length* <= *length*.

22.5.3.1. Errors

ECHANNEL	Invalid domain *channel*
EINVAL	Invalid *flags* value
EBADALIGN	Badly aligned *raddr*, *length*, or *cookie*.
ENORADDR	Bad real address range for local buffer
ENOMAP	*cookie* refers to an invalid map table entry on the exporting side
ENOACCESS	Requested copy operation is not permitted by exporter's map table entry
EBADPGSZ	Page size of *cookie* does not match page size specified in the map table entry
EWOULDBLOCK	Operation would block

22.5.4. ldc_mapin

trap#	FAST_TRAP
function#	LDC_MAPIN
arg0	channel
arg1	cookie
ret0	status
ret1	raddr
ret2	perms

This API service attempts to map into the local guest's real address space the page identified by the shared memory *cookie*. Upon success the service returns the real address the page was mapped at in *raddr*, and the access permissions granted to that page by the exporter for cpu and IO access in *perms*. Bit 0 in the *perms* value corresponds to bit 4 in the export table entry — namely CPU read permission. Bit 1

in *perms* corresponds to bit 5 in the export map table entry, and so on. Bits 5 through 63 of *perms* are undefined and should be ignored.

22.5.4.1. Errors

ECHANNEL	Invalid domain *channel*
EINVAL	Invalid *flags* value
EBADALIGN	Badly aligned *cookie*
ENOMAP	*cookie* refers to an invalid map table entry on the exporting side
ENOACCESS	Requested map operation is not permitted by exporter's map table entry
EBADPGSZ	Page size of *cookie* does not match page size specified in the map table entry
ETOOMANY	Too many mapins already exist
EWOULDBLOCK	Operation would block

22.5.5. ldc_unmap

trap#	FAST_TRAP
function#	LDC_UNMAP
arg0	raddr
ret0	status

This API service attempts to unmap from the local guest's real address space the imported page mapped at the real address *raddr*.

This API may fail if the guest has not already removed any virtual or IOMMU mappings associated with this page.

22.5.5.1. Errors

ENORADDR	Illegal *raddr* value
EBADALIGN	Badly aligned *raddr*
ENOMAP	*raddr* refers to a non-existent imported page
EWOULDBLOCK	Operation would block

22.5.6. ldc_revoke

trap#	FAST_TRAP
function#	LDC_REVOKE
arg0	channel
arg1	cookie
arg2	revoke_cookie
ret0	status

This API service attempts to forcibly unmap from a remote guest's real address space a page previously exported by the local guest. The remote guest is the peer on the other end of the LDC channel specified by *channel*. The *cookie* is the cookie originally passed to that remote guest, the *revoke_cookie* is the revocation cookie supplied by the hypervisor to assist with this API call. This unmapping mechanism

also forcibly unmaps any virtual or IOMMU mappings that the remote guest may be using corresponding to this exported page.

Note

As an optimization, in the event that this API fails with EWOULDBLOCK, the caller should re-read the revocation cookie from the corresponding export table entry; in the event that the revocation cookie has been set to zero, this API should no longer be necessary.

22.5.6.1. Errors

ECHANNEL	Invalid domain *`channel`*
EINVAL	Invalid *`revoke_cookie`*
EBADALIGN	Badly aligned *`cookie`*
EWOULDBLOCK	Operation would block

Chapter 23. PCI I/O Services

23.1. Introduction

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

23.1.1. External documents

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

- sun4v Bus Binding to Open Firmware [http://arc.opensolaris.org/caselog/FWARC/2005/111] ([sun4vbind])
- vPCI Bus Binding to Open Firmware
- PCI Express Base Specification 1.0a [http://www.pcisig.com/specifications/pciexpress/base/archive/] ([pcie2002])

23.2. 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”.

23.3. PCI IO Data Definitions

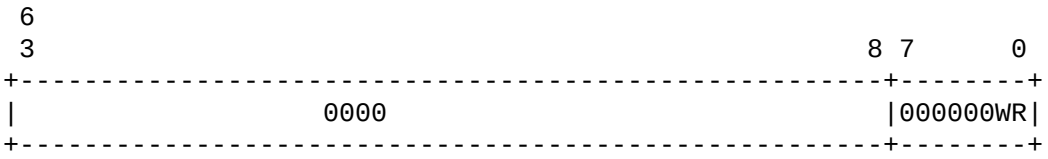
devhandle
: Device handle. The device handle uniquely identifies a sun4v device. It consists of the the lower 28-bits of the hi-cell of the first entry of the sun4v device's "reg" property as defined by the Sun4v Bus Binding to Open Firmware.

tsbnum
: TSB Number. Identifies which IOTSB 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.

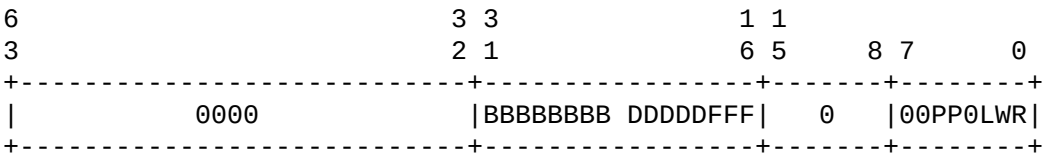
io_attributes
: IO Attributes for IOMMU mappings. Attributes for IOMMU mappings. One or more of the following attribute bits stored in a 64-bit unsigned int.



PCI_MAP_ATTR_READ	0x01	Transfer direction is from memory
PCI_MAP_ATTR_WRITE	0x02	Transfer direction is to memory

Bits 63:2 are unused and must be set to zero for version 1.0 of the specification.

Version 1.1 io_attributes:



The additional IO attributes in version 1.1:

PCI_MAP_ATTR_ RELAXED_ORDERING	0x04	Requested DMA transaction can be relaxed ordered within the root complex (RC).
PCI_MAP_ATTR_ PHANTOM_FUNCTION	bits <5:4>	Value of PCI Express and PCI-X phantom function configuration. Its encoding is identical t the “Phantom Function Supported” field of the “Device Capabilities Register”, offset 0x4 in the “PCI Express Capability Structure”. The structure is part of the device's config space.
PCI_MAP_ATTR_BDF	bits <31:16>	Bus, device and function number of the device that is going to issue DMA transactions. The BDF values are used to guarantee the mapping is only accessed by the specified device. If the BDF is set to all zeros, BDF-based (Requester ID based) protection will be disabled for the mapping.

Relaxed Ordering (`L`) is advisory. Not all hardware implements relaxed ordering. If the relaxed ordering attribute is not implemented in hardware, the implementation is permitted to ignore the Relaxed Ordering attribute.

Version 1.0 Reserved io_attribute bits

For compatibility with future versions of this specification, the caller must set 63:2 to zero.

Version 1.1 Reserved io_attribute bits

For compatibility with future versions of this specification, the caller must set bits 3, 15:16, and 63:32 to zero.

Note

For compatibility with existing hardware and guest behavior, the *R* (Read) bit is implied for all valid mappings. Future versions of this specification may change this behavior by requiring the *R* bit to be set for any mapping intended to be readable by the device.

Note

Note: Some hardware implementations do not implement an *R* (Read) bit in the hardware. In this case, the *R* attribute is implied by any valid IOMMU mapping, and it is not possible to create a write-only mapping. In this case, and for legacy guest support, `pci_iommu_getmap` may return the *R* io_attribute Set even if it wasn't Set when `pci_iommu_map` was called to create that mapping.

r_addr
A 64-bit Real Address.

pci_device
A 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
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. The 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	An 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. Each io_page_address is an r_addr.

io_page_list_p
: A pointer to an io_page_list. io_page_list_p is an r_addr.

"size-based byte swap"
: Some functions do size-based byte swapping which allows software to access pointers and counters in native form when the processor operates in a different endianness than the I/O 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	0x01020304.05060708	0x08070605.04030201

23.4. API calls

The following APIs are provided for PCI services.

23.4.1. 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.

23.4.1.1. Errors

EINVAL	Invalid *devhandle*, tsbnum, tsbindex, io_attributes
EBADALIGN	*r_addr* is not correctly aligned
ENORADDR	*io_page_list_p* is not a valid real address or an entry in the io_page_list is not a valid real address

23.4.2. 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_demapped* 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.

23.4.2.1. Errors

EINVAL	Invalid *devhandle*, tsbnum, tsbindex

23.4.3. 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.

23.4.3.1. Errors

EINVAL	Invalid *devhandle*, tsbnum, tsbindex
ENOMAP	Mapping is not valid, no translation exists

23.4.4. pci_iommu_getbypass

trap#	FAST_TRAP
function#	PCI_IOMMU_GETBYPASS
arg0	devhandle
arg1	r_addr
arg2	io_attributes
ret0	status
ret1	io_address

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.

23.4.4.1. Errors

EINVAL	Invalid *devhandle*, io_attributes
ENORADDR	*r_addr* is not a valid real address
ENOTSUPPORTED	Function is not supported in this implementation

23.4.5. 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 adapter 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.

23.4.5.1. Errors

EINVAL	Invalid *devhandle*, *pci_device*, *offset*, *size*
EBADALIGN	*pci_config_offset* is not size-aligned
ENOACCESS	Access to this offset is not permitted
EWOULDBLOCK	io domain not ready for config space access (See Section 23.5.2.1, "pci_iov_root_configured"

23.4.6. 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 adapter 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.

23.4.6.1. Errors

EINVAL	Invalid *devhandle*, *pci_device*, *offset*, *size*
EBADALIGN	*pci_config_offset* is not size-aligned
ENOACCESS	Access to this offset is not permitted
EWOULDBLOCK	io domain not ready for config space access (See Section 23.5.2.1, “pci_iov_root_configured”

23.4.7. 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.

23.4.7.1. Errors

EINVAL	Invalid *devhandle* or size
EBADALIGN	*r_addr* is not correctly aligned
ENORADDR	*r_addr* is not a valid real address
ENOACCESS	Access to this offset is not permitted

23.4.8. 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.

23.4.8.1. Errors

EINVAL	Invalid *devhandle*, pci_device, or size
EBADALIGN	*r_addr* is not correctly aligned
ENORADDR	*r_addr* is not a valid real address
ENOACCESS	Access to this offset is not permitted
ENOTSUPPORTED	Function is not supported in this implementation

23.4.9. pci_dma_sync

trap#	FAST_TRAP
function#	PCI_DMA_SYNC
arg0	devhandle
arg1	r_addr
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.

23.4.9.1. Errors

EINVAL	Invalid *devhandle* or *io_sync_direction*
ENORADDR	*r_addr* is not a valid real address

23.5. Static Direct I/O

23.5.1. SDIO Definitions

root domain	A domain that owns configuration and mangement of a sun4v pci virtual root complex.
io domain	A domain that has access to devices *below* a sun4v pci virtual root complex but is not the root domain.

23.5.2. SDIO API Definitions

23.5.2.1. pci_iov_root_configured

trap#	FAST_TRAP
function#	PCI_IOV_ROOT_CONFIGURED
arg0	devhandle
ret0	status

The root complex identified by *devhandle* is ready to be shared. The root domain guest calls this when it's ready for other io guests to begin using shared devices under the root complex identified by the argument *devhandle*, which must be the devhandle of a root complex device node owned by this guest.

This call is an indication to the hypervisor that any other guest sharing the devices under this root complex may now access the configuration space of those devices.

In any io guest domain, pci_config_get and pci_config_put shall return EWOULDBLOCK on any attempted access to config space under the root complex defined by the argument *devhandle* until this API is called in the root domain. If the root domain is reset, behavior reverts back to the initial behavior until this API is called in the root domain.

23.5.2.1.1. Errors

EINVAL	Invalid *devhandle*
ENOACCESS	No access, the guest is not the root domain for this root complex *devhandle*

23.5.2.2. pci_real_config_get

trap#	FAST_TRAP
function#	PCI_REAL_CONFIG_GET
arg0	devhandle
arg1	pci_device
arg2	pci_config_offset
arg3	size
ret0	status
ret1	error_flag
ret2	data

Read the real PCI configuration space for the PCI adapter 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.

23.5.2.2.1. Errors

EINVAL	Invalid *devhandle*, pci_device, offset, size
EBADALIGN	*pci_config_offset* is not size-aligned
ENOACCESS	Access to this offset is not permitted or not the root domain for this root complex *devhandle*

23.5.2.3. pci_real_config_put

trap#	FAST_TRAP
function#	PCI_REAL_CONFIG_PUT
arg0	devhandle
arg1	pci_device
arg2	pci_config_offset
arg3	size
arg4	data
ret0	status
ret1	error_flag

Write real PCI config space for the PCI adapter 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.

23.5.2.3.1. Errors

EINVAL	Invalid *devhandle*, *pci_device*, *offset*, *size*
EBADALIGN	*pci_config_offset* is not size-aligned
ENOACCESS	Access to this offset is not permitted or not the root domain for this root complex *devhandle*

23.5.2.4. pci_error_send

trap#	FAST_TRAP
function#	PCI_ERROR_SEND
arg0	devhandle
arg1	devino
arg2	pci_device
ret0	status

Send an error packet to any io guest sharing the device identified by the argument *pci_device* in the fabric identified by the argument *devhandle*.

devhandle must be the devhandle of a PCI root complex, owned by this guest. *devino* must be the devino associated with pci error packets for this root complex.

pci_device is the device reporting the error. If *pci_device* is zero, all other guests sharing this pci fabric will receive an error packet. If *pci_device* is non-zero, only those guests sharing the device identified by the *pci_device* argument will receive the error packet.

The error packet is delivered to the dev mondo queue of the io guest sharing this device, if any. The first entry, *SYSINO*, will be the correct value for the *devhandle*, *devino* in that guest that represents the shared root complex, identified by the arguments *devhandle*, *devino* for this guest.

Note

The epkt is never delivered to the domain that called this API.

Refer to [vpcierrs] for definition of the contents of pci error packets.

The packet shall contain the following fields:

0x00	*sysino*	devhandle, devino of io guest's root complex
0x08	*ehdl*	unique error handle, generated by hypervisor

0x10	*stick*	timestamp, the value of the %stick register
0x18	*desc*	see below, error-specific
0x20	0	
0x28	0	
0x30	0	
0x38	0	

The DESC field shall contain the following values:

```
31:28 - Block - value 1 - hostbus
27:24 - Op - Value 0
23:20 - Phase - Value 0
19:16 - Cond - Value 0
15:12 - Dir - value 0
 11:0 - Flags, bit 11 (STOP) Set, everything else Clear.
```

23.5.2.4.1. Errors

EINVAL	Invalid *devhandle*, *pci_device*, or *devino*
ENOACCESS	Not the root domain for this root complex *devhandle*

Chapter 24. PCI MSI Services

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

24.1. 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.

24.2. 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 [sun4vbind]. 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 [sun4vbind].

Each MSI EQ is compliant with the definition of interrupt queues described in [pcie2002], 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.

24.2.1. 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:

```
6666555555555544444444443333333333222222222211111111110000000000
3210987654321098765432109876543210987654321098765432109876543210
0x00:
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVxxxxxxxxxxxxxxxxxxxxxxxxTTTTTTTT
0x08:
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
0x10:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0x18:
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
0x20:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxRRRRRRRRRRRRRRR
0x28:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0x30:
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
0x38:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Where,

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

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. The upper 4 bits are reserved, and must be zero.

0000	Not an MSI data record, reserved for software use
0001	MSG
0010	MSI32
0011	MSI64
0100	Reserved
...	Reserved
0111	Reserved
1000	INTx
1001	Reserved
...	Reserved
1110	Reserved
1111	Not an MSI data record, reserved for software use

All other encodings are reserved.

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

SS..SS is the message time-stamp 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.ddddd fff, 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 device ID, and *fff* is the target function number.

CCC is the message routing code as defined by [pcie2002]

MM..MM is the message code as defined by [pcie2002]

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

00	INTA
01	INTB
02	INTC
03	INTD

24.3. 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	MSI32
1	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:

0	PCI_MSIQSTATE_IDLE (idle, non-error state)
1	PCI_MSIQSTATE_ERROR (error state)

msiqvalid — An unsigned integer containing one of the following values:

0	PCI_MSIQ_INVALID (disabled/invalid)
1	PCI_MSIQ_VALID (enabled/valid)

msistate — An unsigned integer containing one of the following values:

0	PCI_MSISTATE_IDLE (idle/not enabled)
1	PCI_MSISTATE_DELIVERED (MSI delivered)

msivalid — An unsigned integer containing one of the following values:

0	PCI_MSI_INVALID (disabled/invalid)
1	PCI_MSI_VALID (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:

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

msgvalid — An unsigned integer containing one of the following values:

0	PCI_MSG_INVALID (disabled/invalid)
1	PCI_MSG_VALID (enabled/valid)

24.4. API calls

24.4.1. 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.

24.4.1.1. Errors

EINVAL	Invalid *devhandle*, *msiqid*, or nentries
ENORADDR	*r_addr* is not a valid real address
EBADALIGN	*r_addr* is not properly aligned

24.4.2. pci_msiq_info

trap#	FAST_TRAP
function#	PCI_MSIQ_INFO
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 zero is returned in *nentries*.

24.4.2.1. Errors

EINVAL	Invalid *devhandle* or *msiqid*

24.4.3. 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*.

24.4.3.1. Errors

EINVAL	Invalid *devhandle* or *msiqid*

24.4.4. 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.

24.4.4.1. Errors

EINVAL	Invalid *devhandle*, *msiqid*, or *msiqvalid* value or the MSI EQ is uninitialized.

24.4.5. 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*.

24.4.5.1. Errors

EINVAL	Invalid *devhandle* or *msiqid*

24.4.6. 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.

24.4.6.1. Errors

EINVAL	Invalid *devhandle*, *msiqid*, or *msiqstate* value or MSI EQ is uninitialized

24.4.7. 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 arguments *devhandle*, *msiqid*.

24.4.7.1. Errors

EINVAL	Invalid *devhandle* or *msiqid* or MSI EQ is uninitialized

24.4.8. pci_msiq_sethead

trap#	FAST_TRAP
function#	PCI_MSIQ_SETHEAD
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.

24.4.8.1. Errors

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

24.4.9. pci_msiq_gettail

trap#	FAST_TRAP

function#	PCI_MSIQ_GETTAIL
arg0	devhandle
arg1	msiqid
ret0	status
arg2	msiqtail

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

24.4.9.1. Errors

EINVAL	Invalid *devhandle* or *msiqid* or MSI EQ is uninitialized

24.4.10. 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*.

24.4.10.1. Errors

EINVAL	Invalid *devhandle* or *msinum*

24.4.11. 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*

24.4.11.1. Errors

EINVAL	Invalid *devhandle*, *msinum*, or *msivalidstate*

24.4.12. 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*.

24.4.12.1. Errors

EINVAL	Invalid *devhandle* or *msinum* or MSI is unbound

24.4.13. pci_msi_setmsiq

trap#	FAST_TRAP
function#	PCI_MSI_SETMSIQ
arg0	devhandle
arg1	msinum
arg2	msiqid
arg3	msitype
ret0	status

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

24.4.13.1. Errors

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

24.4.14. 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.

24.4.14.1. Errors

EINVAL	Invalid *devhandle* or *msinum*

24.4.15. 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*.

24.4.15.1. Errors

EINVAL	Invalid *devhandle*, *msinum*, or *msistate*

24.4.16. 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*.

24.4.16.1. Errors

EINVAL	Invalid *devhandle* or *msgtype*

24.4.17. pci_msg_setmsiq

trap#	FAST_TRAP
function#	PCI_MSG_SETMSIQ
arg0	devhandle
arg1	msgtype
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*.

24.4.17.1. Errors

EINVAL	Invalid *devhandle*, *msgtype*, or *msiqid*

24.4.18. 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*.

24.4.18.1. Errors

EINVAL	Invalid *devhandle* or *msgtype*

24.4.19. 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*, *msgtype* to the valid/enabled state defined by the argument *msgvalidstate*

24.4.19.1. Errors

EINVAL	Invalid *devhandle*, *msgtype*, or *msgvalidstate*

Chapter 25. Cryptographic services

The following APIs provide access via the Hypervisor to hardware assisted cryptographic functionality. These APIs may only be provided by certain platforms, and even then may not be available to all virtual machines. Restrictions on the use of these APIs may be imposed in order to support live-migration and other system management activities.

25.1. Random Number Generation

The UltraSPARC-T2 incorporates a hardware random number generator to support cryptographic functionality. This provides a source of entropy to be used by Operating System cryptographic frameworks to ultimately provide efficient random number generation to higher layers of software.

The random number generation (RNG) APIs provide two forms of access to the underlying RNG hardware; configuration & management, and random number data access.

25.1.1. Trusted Domains

In order to provide system-wide security, the configuration & management APIs are restricted in multiple domain configurations to use only by Trusted Domains, for example the Control Domain.

Only Trusted domains are allowed configuration and diagnostic control of the RNG.

Trusted domains are designated by the LDom manager with enforcement of such designation implemented within the Hypervisor. Attempts by a non-trusted domain to access Control or Diagnostic related API entry points will fail with ENOACCESS errors.

Note that access to Control and Diagnostic entry points is dynamic and can be taken away at anytime from a domain. Exactly one (1) domain must exist as the Trusted Domain to ensure proper RNG behavior.

In the 1.0 API, trusted domains were able to execute the `rng_get_diag_control` call, which gave them exclusive access to the trusted domain API calls. In the 2.0 API, this call has been dropped and only one domain, as specified by the LDoms manager, is allowed access to the trusted domain API calls.

The RNG operations are restricted as follows for the 1.0 API:

Trusted Domain(s)	**Any Domain**
RNG_GETDIAG_CONTROL	RNG_DATA_READ
RNG_CTL_READ	
RNG_CTL_WRITE	
RNG_DATA_READ_DIAG	

The RNG operations are restricted as follows for the 2.0 API:

Trusted Domain(s)	**Any Domain**
RNG_CTL_READ	RNG_DATA_READ
RNG_CTL_WRITE	
RNG_DATA_READ_DIAG	

25.1.2. RNG Control Register data structure

Each RNG has both a read-only data register and a control register. The control register is used to modify the behavior of the random number generator, while the data register contains each of the random numbers consumed by client software. Both registers are 64-bits wide.

The RNG generates random numbers by using three independent noise generators. Each of these noise generators may be individually programmed by writing to the same RNG control register a bit string in bits 3:0 that specifies which noise generator needs to be programmed. If none of the bits are set, the noise cells are turned off. If any combination of two or more bits are set, all of the noise generators are selected. In this way, there are four ways in which the noise generator apparatus may be programmed. One can select noise cell 1, cell 2, cell 3, or all of the cells at once.

Due to this noise generator selection scheme, four successive writes to the same control register with different settings in bits 3:0 will completely configure the RNG.

To expedite writing out the control register's four possible settings, the control register read and write APIs use a data structure comprised of all four possible control register settings.

Offset	Size	Field	Description
0	8	*rng_ctl1*	Control bit for cell selection 1
8	8	*rng_ctl2*	Control bit for cell selection 2
16	8	*rng_ctl3*	Control bit for cell selection 3
24	8	*rng_ctl4*	Control bit for cell selection 4

25.1.3. RNG State

Specifies the state of the RNG and is set during `rng_ctl_write` operations.

Table 25.1. RNG states

Name	Value
RNG_STATE_UNCONFIGURED	0
RNG_STATE_CONFIGURED	1
RNG_STATE_HEALTHCHECK	2
RNG_STATE_ERROR	3

When “configured” the RNG is available for general `rng_data_read` operations. In “health check” mode the RNG is generally unavailable and assumed to be going through a health check sequence via a Trusted domain. Once the health check is complete the Trusted domain will return the RNG to a “configured” state. If the health check determines that the RNG is faulty then it will be left in the “error” state and thus unavailable for any `rng_data_read` operations.

25.1.4. Maximum Data Read Length

The minimum length in bytes that can be read from the hardware RNG Data Register by `rng_data_read_diag` is defined to be 8 bytes.

The maximum length in bytes that can be read from the hardware RNG Data Register by `rng_data_read_diag` is defined to be 128K bytes (128 * 1024).

25.1.5. RNG Mutual Exclusion

All of the RNG hypervisor entry points are protected through mutual exclusion by the hypervisor to ensure that only one thread of control is operating on the RNG at a time. This is necessary to prevent against competing threads (or OS Guests) from re-initializing the RNG hardware while a Data read is possibly in progress from another thread.

The hypervisor does not block waiting for access to the RNG device, instead it will return to the caller with a EWOULDBLOCK error indicating that the hardware device was temporarily unavailable.

25.1.6. RNG Data Availability

The hypervisor will return EWOULDBLOCK errors when attempts are made to read data when no random data is available.

25.1.7. RNG Watchdog Timeout

Each RNG device may be configured for a specific amount of time. Once that timeout value is reached, the RNG device is transitioned to the “unconfigured” state. The watchdog delta is the number of system ticks until the RNG transfers to the “unconfigured” state.

25.1.8. rng_data_read

trap#	FAST_TRAP
function#	RNG_DATA_READ
arg0	*raddr*
ret0	*status*
ret1	*delta*

API for reading a single 64-bit quantity from the RNG Data Register. The contents of the RNG Data Register are stored into the buffer specified by *raddr*. The buffer address must be a real address and 8-byte aligned.

The RNG register must be in the RNG_STATE_CONFIGURED state in order to successfully read from the Data Register.

The client specifies the RNG to which it wants access by providing the *rng_id* (*arg3*) associated with that RNG. If the *rng_id* is invalid then EINVAL is returned.

If this function returns EWOULDBLOCK, indicating that the hardware isn't ready to respond to the request, then it also returns a system clock tick value in *ready_delta* indicating how many system clock ticks before the RNG will be available for a subsequent operation. Note that it is also possible for the caller to encounter an EWOULDBLOCK error should another thread simply be doing a RNG operation at the same time. In this situation the returned *ready_delta* will likely be 0 indicating that the RNG is immediately available for retrying.

25.1.8.1. Errors

EIO	RNG is currently Unconfigured or in a Healthcheck
ENOACCESS	RNG is in the Error state and unavailable
EBADALIGN	Pointer address is not correctly aligned
ENORADDR	Pointer address is not a valid real address
EWOULDBLOCK	RNG currently in use by another thread or it has not yet reached its steady state

25.1.9. rng_ctl_read (2.0)

trap#	FAST_TRAP
function#	RNG_CTL_READ

arg0	*raddr*
arg1	*rng_id*
ret0	*status*
ret1	*state*
ret2	*ready_delta*
ret3	*watchdog_delta*
ret4	*write_status*

This API will store the contents of the RNG control register into the structure pointed to by *raddr* (*arg0*). This address must be a real address, physically contiguous, and aligned on an 8-byte boundary. If *arg0* is NULL (0), then no control register information will be stored. This API will also return the current *state* (*ret1*), the current *ready_delta* (*ret2*), which specifies in how many system clock ticks from the present time the RNG data register will be available for reading, the *watchdog delta* (*ret3*), which specifies in how many system clock ticks from the present time the RNG will transition to an error state, and the *write_status** (*ret4*), which provides the status of the last write operation on the control register for this RNG.

If the *ready_delta* has a value of zero, it indicates that the RNG data register is immediately available.

If *watchdog_delta* has a value of zero, it means that either the RNG has transitioned to the *unconfigured* state or that it was set initially to keep its current state in perpetuity. The client is responsible for checking *state* to see in which way the watchdog timeout was used. If *state* refers to either the *healthcheck* or *configured* states, it means that the watchdog timeout was not set. If *state* refers to the *unconfigured* state, it means that the RNG needs to be re-programmed.

The client specifies the RNG to which it wants access by providing the *rng_id* (*arg1*) associated with that RNG. If the *rng_id* is invalid an error is returned.

It is possible for the caller to encounter an EWOULDBLOCK error should another thread be doing a RNG operation on the same *rng_id* at the same time. If the RNG currently has a write operation pending, an EBUSY error will be returned. If Hypervisor returns an EBUSY error, the value in *write_status* is undefined.

25.1.9.1. Programming note

The actual N2 RNG hardware control register does not return the same contents that were written from a previous write operation. Thus, the Hypervisor will keep a snapshot of what was written on a previous `rng_ctl_write` and simply returns this information whenever `rng_ctl_read` is called.

25.1.9.2. Write status

EOK	Success
EIO	Last write operation failed, the RNG contains the last valid configuration

25.1.9.3. Errors

EBADALIGN	Pointer address is not correctly aligned
EINVAL	Invalid RNG ID
EBUSY	RNG has a pending write operation
ENORADDR	Pointer address is not a valid real address

EWOULDBLOCK	RNG currently in use by another thread, caller should retry
ENOACCESS	Caller does not have permission to call this API

25.1.10. rng_ctl_write (2.0)

trap#	FAST_TRAP
function#	RNG_CTL_WRITE
arg0	*raddr*
arg1	*newstate*
arg2	*watchdog_timeout*
arg3	*rng_id*
ret0	*status*

This is the API for initializing the RNG hardware by writing to the RNG control register with the contents of the structure pointed to by *raddr* (*arg0*). Write operations are asynchronous. Before returning to the caller, the hypervisor will schedule the control register to be written out at a future time. If the caller attempts to write to the same RNG before the prior write attempt completes, this call will return with an EBUSY error number.

raddr is a real address and must be physically contiguous and aligned on an 8-byte boundary. The state of the RNG will be set to *newstate* (*arg1*) and must be one of the state values specified in Section 25.1.3, “RNG State”.

The ready delta of the RNG will be determined by finding the largest wait counter value from the control register settings specified by *raddr*. That value will be used by Hypervisor to monitor reads of the RNG data register. EWOULDBLOCK will be returned when the pool is empty and reads of the data register are attempted before the wait counter has been exhausted.

When setting the state to RNG_STATE_CONFIGURED the caller also specifies a *watchdog_timeout* (*arg2*), in system ticks (delta from the current time), to indicate when the current configuration setting will effectively expire. Once this time has expired the RNG will be placed into the RNG_STATE_UNCONFIGURED state thus taking the RNG out of the pool for data reads. Specifying a *watchdog_timeout* value of zero (0) disables the watchdog timeout for the new configuration setting. The intent of providing a timeout is to allow a Trusted Guest to enforce a policy of periodic “health checks” of the RNG hardware if required. The *watchdog_timeout* argument is ignored when specifying any state other than RNG_STATE_CONFIGURED.

For the sake of backwards compatibility, there will be a watchdog timeout threshold. If the client sets a value at or below that threshold, the value is assumed to be zero. The threshold is 60 seconds worth of system ticks.

The client specifies the RNG to which it wants access by providing the *rng_id* (*arg3*) associated with that RNG. If the *rng_id* is invalid then EINVAL is returned.

It is possible for the caller to encounter an EWOULDBLOCK error should another thread be doing a RNG operation on the same RNG at the same time.

25.1.10.1. Errors

EIO	Write to RNG control register failed
EBUSY	A write operation for the specific RNG is still pending
EINVAL	Specified state is not a valid value or an invalid RNG ID was specified

EBADALIGN	Pointer address is not correctly aligned
ENORADDR	Pointer address is not a valid real address
EWOULDBLOCK	RNG currently in use by another thread
ENOACCESS	Caller does not have permission to call this API

25.1.11. rng_data_read_diag (2.0)

trap#	FAST_TRAP
function#	RNG_DATA_READ_DIAG
arg0	*raddr*
arg1	*size*
arg2	*rng_id*
ret0	*status*
ret1	*ready_delta*

This API provides access to 64-bit quantities from the RNG Data Register. The contents of the RNG Data Register are repeatedly read and stored into consecutive locations starting at the specified *raddr*.

The buffer address in *raddr* must be a real address, size aligned, and physically contiguous. The buffer *size* specifies the size of the buffer in bytes and must be a multiple of 8. If the size is greater than 8 then the RNG Data Register will be re-read into consecutive locations in the buffer for each multiple of 8 specified by size. For example, if a buffer size of 32 is specified then the RNG Data Register will be read 4 times (32/8) with each read consecutively stored into the buffer address.

If this function returns EWOULDBLOCK, indicating that the hardware isn't ready to respond to the request, then it also returns a system clock tick value in *ready_delta* indicating how many system clock ticks from the current time that the RNG will be available for a subsequent operation.

25.1.11.1. Programming Note

The caller of this API must have Diagnostic Control of the RNG in order to invoke this operation (see Section 25.1.12.1, “rng_get_diag_control (1.0)”).

25.1.11.2. Errors

EINVAL	Specified buffer size is not valid or an invalid RNG ID has been specified
EBADALIGN	Pointer address is not correctly aligned
ENORADDR	Pointer address is not a valid real address
EWOULDBLOCK	RNG is currently in use by another thread or it has not yet reached its steady state. The caller should retry in *ready_delta* system clock ticks.
ENOACCESS	Caller does not have permission to call this API

25.1.12. Deprecated RNG 1.0 APIs

25.1.12.1. rng_get_diag_control (1.0)

trap#	FAST_TRAP
function#	RNG_GET_DIAG_CONTROL

ret0	*status*

This API gives the calling Guest OS diagnostic control over the RNG for performing subsequent `rng_ctl_write` and `rng_data_read_diag` operations. Only one Guest at a time is permitted to execute the aforementioned diagnostic operations. Control will remain with the current Guest until another Guest takes control by invoking this same entry point.

25.1.12.1.1. Errors

EWOULDBLOCK	RNG currently in use by another thread
ENOACCESS	Caller does not have permission to call this API

25.1.12.2. rng_ctl_read (1.0)

trap#	FAST_TRAP
function#	RNG_CTL_READ
arg0	*raddr*
ret0	*status*
ret1	*state*
ret2	*delta*

This API will store the contents of the RNG Control registers into the RNG control structure pointed to by *raddr*. This address must be a real address, physically contiguous, and aligned on an 8-byte boundary. If *raddr* is NULL (0), then no Control register information will be stored. This API will also return the current *state*, and the current ready *delta* which specifies how many system clock ticks from the present time that the RNG will be available for further operations. A value of zero indicates that the RNG is immediately available.

25.1.12.2.1. Programming note

The actual N2 RNG hardware control register does not return the same contents that were written from a previous write operation. Thus, the Hypervisor will keep a snapshot of what was written on a previous `rng_ctl_write` and simply return this information whenever `rng_ctl_read` is called.

25.1.12.2.2. Errors

EBADALIGN	Pointer address is not correctly aligned
ENORADDR	Pointer address is not a valid real address
EWOULDBLOCK	RNG currently in use by another thread, caller should retry
ENOACCESS	Caller does not have permission to call this API

25.1.12.3. rng_ctl_write (1.0)

trap#	FAST_TRAP
function#	RNG_CTL_WRITE
arg0	*raddr*
arg1	*newstate*
arg2	*timeout*
ret0	*status*

ret1	*delta*

This API is used to initialize the RNG hardware by writing to the RNG Control register with the contents of the structure pointed to by *raddr*. This address must be a real address, physically contiguous, and aligned on an 8-byte boundary. The state of the RNG will be set to *newstate* and must be one of the state values specified in Section 25.1.3, “RNG State”.

When setting the state to RNG_STATE_CONFIGURED the caller also specifies a *timeout*, in system ticks (a delta from the current time), to indicate when the current configuration setting will effectively expire. Once this time has expired the hypervisor will put the RNG into the RNG_STATE_ERROR state thus making the RNG unavailable for Data Reads. A *timeout* value of zero (0) indicates an infinite lifetime for the new configuration setting.

If this function returns EWOULDBLOCK, indicating that the hardware isn't ready to respond to the request, it also returns a value in delta (in system clock ticks) indicating when the RNG will be available for a subsequent operation. This delay in having the RNG available occurs after a previous rng_ctl_write operation and is to allow the RNG to reach a steady state after it has been configured.

25.1.12.3.1. Programming note

The intent of providing a timeout is to allow a Trusted Guest to enforce a policy of periodic “health checks” of the RNG hardware if required. The timeout argument is ignored when specifying any state other than RNG_STATE_CONFIGURED.

Note also that the caller must have Diagnostic Control of the RNG in order to invoke this operation (see Section 25.1.12.1, “rng_get_diag_control (1.0)”).

25.1.12.3.2. Errors

EIO	The calling Guest does not currently have Diagnostic Control to manipulate the RNG settings. Caller must first invoke rng_get_diag_control.
EINVAL	Specified state is not a valid value
EBADALIGN	Pointer address is not correctly aligned
ENORADDR	Pointer address is not a valid real address
EWOULDBLOCK	RNG currently in use by another thread or it has not yet reached its steady state, caller should retry in delta clock ticks.
ENOACCESS	Caller does not have permission to call this API

25.1.12.4. rng_data_read_diag (1.0)

trap#	FAST_TRAP
function#	RNG_DATA_READ_DIAG
arg0	*raddr*
arg1	*size*
ret0	*status*
ret1	*delta*

This API provides access to 64-bit quantities from the RNG Data Register. The contents of the RNG Data Register are repeatedly read and stored into consecutive locations starting at the specified *raddr*. The buffer address in *raddr* must be a real address, size aligned, and physically contiguous. The buffer *size* specifies the size of the buffer in bytes and must be a multiple of 8.

If the size is greater than 8 then the RNG Data Register will be re-read into consecutive locations in the buffer for each multiple of 8 specified by size. For example, if a buffer size of 32 is specified then the RNG Data Register will be read 4 times (32/8) with each read consecutively stored into the buffer address.

If this function returns EWOULDBLOCK, indicating that the hardware isn't ready to respond to the request, then it also returns a system clock tick value in *delta* indicating how many system clock ticks from the current time that the RNG will be available for a subsequent operation.

25.1.12.4.1. Programming Note

The caller of this API must have Diagnostic Control of the RNG in order to invoke this operation (see Section 25.1.12.1, “rng_get_diag_control (1.0)”).

25.1.12.4.2. Errors

EIO	The calling Guest does not currently have Diagnostic Control to manipulate the RNG settings. Caller must first invoke `rng_get_diag_control`.
EINVAL	Specified buffer size is not valid
EBADALIGN	Pointer address is not correctly aligned
ENORADDR	Pointer address is not a valid real address
EWOULDBLOCK	RNG currently blocked
ENOACCESS	Caller does not have permission to call this API

25.2. Niagara crypto services

This sections describes the Niagara Crypto Service (NCS) Hypervisor API for the UltraSPARC-T1 and UltraSPARC-T2 processors. This API is designed to resemble the queuing interfaces provided by other hypervisor APIs.

This interface is designed to be used by a more generic cryptographic framework provided by a guest Operating System. (For example the Solaris Cryptographic Framework). Therefore these hypervisor services only provide access to chip specific functionality, rather than providing more generic cryptographic operations.

25.2.1. Versioning

The interface presented here represents version 2.0. The previous NCS hypervisor API representing version 1.x is now deprecated.

25.2.2. Work queues

The UltraSPARC-T1 processor provides a multiply-accumulate-unit associated with each processor core to be used for accelerating bulk cryptographic operations. UltraSPARC-T2 extended this functionality and added a random number generator, and support for more advanced cryptographic operations via the CWQ. A full description of this functionality can be found in the programmer's reference manuals for these chips, and so is not discussed further here.

Work is submitted to a cryptographic unit via a queue, and similarly results are enqueued by the hypervisor upon completion. A queue type parameter is used to select between MAU and CWQ functionality for work submission.

The queues are managed as circular arrays with head and tail pointers indicating where active jobs are present— Operation of the queues is analogous to the interrupt queues. Note: Byte ordering of all fields is Big-endian.

25.2.2.1. Queue Type

The queue type parameter specifies whether the queue being operated on represents either the MAU or CWQ, and has one of the values as specified below:

Table 25.2. Niagara Crypto queue types

NCS_QTYPE_MAU	0x01
NCS_QTYPE_CWQ	0x02 (UltraSPARC-T2 only)

The queue handle parameter specifies a 64-bit unsigned integer value that uniquely identifies the queue being operated on.

25.2.2.2. MAU queue

The MAU queue is described by an array of 64-byte entries where each entry is described by the following structure:

Table 25.3. Niagara Crypto MAU queue entry

Offset	Size	Name	Description
0	8	*nhd_state*	State Valid values: ND_STATE_FREE 0 Entry is unused ND_STATE_PENDING 1 Allocated and pending submission to MAU ND_STATE_BUSY 2 Entry has been submitted to MAU ND_STATE_DONE 3 Entry has been successfully executed ND_STATE_ERROR 4 Entry completed execution, with an error
8	8	*nhd_type*	Bit flags to delineate independent MAU jobs which may be comprised of one or more queue entries. Interrupts are only sent to the OS when the Last entry in a job has been completed. Valid values: ND_TYPE_UNASSIGNED 0x00 Entry is unused ND_TYPE_START 0x01 Entry is the start of a job ND_TYPE_CONT 0x02 Continuation of an existing job ND_TYPE_END 0x80 Entry is the end of a job
16	32	*nhd_regs*	Values to be installed in the MAU hardware registers. The *nhd_regs* field is a 32 byte structure with the following format: Offset | Size | Name | Description 0 | 8 | *mr_ctl* | MA Control Register 8 | 8 | *mr_mpa* | MA Physical Address Register

<table>
<tr><th>Offset</th><th>Size</th><th>Name</th><th>Description</th></tr>
<tr><td></td><td></td><td></td><td>
<table>
<tr><th>Offset</th><th>Size</th><th>Name</th><th>Description</th></tr>
<tr><td>16</td><td>8</td><td>mr_ma</td><td>MA Memory Address Register</td></tr>
<tr><td>24</td><td>8</td><td>mr_np</td><td>MA NP Register</td></tr>
</table>
The exact definition of these registers is given in the Programmer's Reference Manual for the UltraSPARC-T1 or UltraSPARC-T2 processors, and is beyond the scope of this document.
</td></tr>
<tr><td>48</td><td>8</td><td>nhd_errstatus</td><td>Bit flags indicating type of MAU error which may have occurred with respect to descriptor.
Valid values:
<table>
<tr><td>ND_ERR_OK</td><td>0x00</td><td>No Error</td></tr>
<tr><td>ND_ERR_INVOP</td><td>0x01</td><td>Invalid MAU operation</td></tr>
<tr><td>ND_ERR_HWE</td><td>0x02</td><td>Hardware error detected by MAU</td></tr>
</table>
</td></tr>
<tr><td>56</td><td>8</td><td>Padding</td><td>Padding out to 64 bytes.</td></tr>
</table>

25.2.2.3. CWQ queue (UltraSPARC-T2 only)

The CWQ queue is described by an array of 64-byte entries where each entry is described by the following structure:

Table 25.4. Niagara Crypto CWQ queue entry

Offset	Size	Name	Description
0	8	*cw_ctlbits*	Control bits indicating the nature of the respective control word.
8	8	*cw_src_addr*	Real address of source data.
16	8	*cw_auth_key_addr*	Real address of location containing authentication key.
24	8	*cw_auth_iv_addr*	Real address of location containing initial value for authentication.
32	8	*cw_final_auth_state_addr*	Real address of the location that will be used to hold the final authentication state.
40	8	*cw_enc_key_addr*	Real address of location containing encryption key.
48	8	*cw_enc_iv_addr*	Real address of location containing encryption initialization vector.
56	8	*cw_dst_addr*	Real address of destination buffer.

The PRM for UltraSPARC-T2 details the exact definition of these fields.

25.2.3. ncs_qconf

trap#	FAST_TRAP
function#	NCS_QCONF

arg0	*queue_type*
arg1	raddr/handle
arg2	*size*
ret0	*status*
ret1	*handle*

This API is used for configuring or unconfiguring either a MAU queue or a CWQ queue as specified by *queue_type* (*arg0*).

The the real address of the base of the queue is given in *raddr* (*arg1*) and must be aligned on a queue size boundary. For example, a 32 entry MAU or CWQ queue must be aligned on a 2048-byte real address boundary. When unconfiguring a queue, the *handle* (*arg1*) represents the queue to be unconfigured.

The number of entries in the queue is given in *size* (*arg2*) and must be a power of 2. A value of zero (0) unconfigures the given queue represented by the queue *handle* (*arg1*).

25.2.3.1. Programming note

Upon success, when configuring a queue, the caller is returned a queue *handle* (*ret1*) which must be used for subsequent queue operations. Note that the queue being configured is only of the MAU/CWQ for the processor core containing the CPU upon which the caller is executing.

The calling thread should bind itself to the current CPU to ensure its context does not get switched to a different CPU and possibly a different core during the operation.

25.2.3.2. Errors

EINVAL	Specified queue type is not recognized, or specified queue size is not a power of 2, or queue handle is invalid
ENOACCESS	CPU does not have access to a MAU/CWQ
EBADALIGN	Base address of queue is not correctly aligned
ENORADDR	Pointer address is not a valid real address

25.2.4. ncs_qinfo

trap#	FAST_TRAP
function#	NCS_QINFO
arg0	*handle*
ret0	*status*
ret1	*type*
ret2	*raddr*
ret3	*size*

This API retrieves the queue *type* and the real address of the base of the queue (in *raddr*), and the queue *size* for the queue identified by the queue *handle* (*arg0*).

25.2.4.1. Errors

EINVAL	Queue handle is invalid

25.2.5. ncs_gethead

trap#	FAST_TRAP
function#	NCS_GETHEAD
arg0	*handle*
ret0	*status*
ret1	*offset*

This API retrieves the head offset for the queue identified by *handle* (*arg0*). The head *offset* represents the current beginning point for queue jobs to be processed. There is no guarantee that subsequent to calling this entry point that the head will not move forward.

25.2.5.1. Errors

EINVAL	Queue handle is invalid

25.2.6. ncs_sethead_marker

trap#	FAST_TRAP
function#	NCS_SETHEAD_MARKER
arg0	*handle*
arg1	*offset*
ret0	*status*

This API tells the hypervisor to set the head *offset* (*arg1*) for a given queue handle (*arg0*) to the specified value. This value is used to effectively determine how far along the caller has processed the queue of descriptors relative to where the CWQ hardware is currently operating. This value is NOT stored into the actual CWQ hardware Head register since that register is managed by hardware once a queue has been configured and enabled.

The *offset* must be aligned on a 64-byte boundary. Any attempt to specify a head offset value that resides after the hardware's notion of the current head and before the hardware's notion of the current tail will result in an EINVAL error.

25.2.6.1. Errors

EINVAL	Queue *handle* is invalid or specified queue head *offset* value is invalid

25.2.7. ncs_gettail

trap#	FAST_TRAP
function#	NCS_GETTAIL
arg0	*handle*
ret0	*status*
ret1	*offset*

This API retrieves the tail *offset* (*ret1*) for the queue identified by the queue *handle* (*arg0*).

The tail represents the current point for enqueuing new jobs. Changes in the tail can only happen via the ncs_settail API.

25.2.7.1. Errors

EINVAL	Queue *handle* is invalid

25.2.8. ncs_settail

trap#	FAST_TRAP
function#	NCS_SETTAIL
arg0	*handle*
arg1	*tail_offset*
ret0	*status*

This API tells the hypervisor to set the *tail_offset* for a given queue *handle* (*arg0*) to the value specified in *offset* (*arg1*). The hypervisor will automatically start processing of operations starting at the current head pointer, if not already in progress.

The offset must be aligned on a 64-byte boundary and calculated so as to increase the number of pending entries on the queue. Any attempt to decrease the number of pending queue entries is considered an invalid tail offset and will result in an EINVAL error.

25.2.8.1. Programming note

Care must be taken with multi-threaded guest code where a scheduler may move the calling thread to another virtual CPU. To ensure that the caller does not get switched to a different CPU and thus possibly a different core and crypto queue between enqueuing a job and calling the ncs_settail API, the caller should bind itself to the target CPU.

The caller can wait for an asynchronous interrupt indicating completion of a job in the queue at which point the caller must check the current head/tail pointers to verify whether their job has completed.

25.2.8.2. Errors

EINVAL	Queue *handle* is invalid or queue *tail_offset* value is invalid
ENORADDR	Buffer address referenced in queue entry is not a valid real address

25.2.9. ncs_qhandle_to_devino

trap#	FAST_TRAP
function#	NCS_QHANDLE_TO_DEVINO
arg0	*handle*
ret0	*status*
ret1	*devino*

This API retrieves the interrupt number (*devino*) for the crypto unit represented by the given queue *handle* (*arg0*).

25.2.9.1. Errors

EINVAL	Queue *handle* is invalid

25.2.10. ncs_ulqconf (version 2.1)

trap#	FAST_TRAP
function#	NCS_ULQCONF
arg0	*base_address* or *queue_handle*
arg1	*page_size* (encoded)
arg2	*queue_size* (#entries)
ret0	*status*
ret1	*queue_handle*

This API is for configuring or unconfiguring the CWQ to set it up for the new User-land mode of operation of UltraSPARC-KT[ua2007n3]. When configuring a queue the caller is returned a *queue_handle* (*ret1*) which can be used to unconfigure the given CWQ.

When configuring a queue, the *base_address* (*arg0*) is the real address of the page that is dedicated for the queue and all data that is used by this CWQ during user-land operation. The queue itself should be set up within the first 4K of the page. The page sizes supported are 4M and 256M as the smaller sizes enabled by the specification (8K and 64K) would be too small for smooth operation.

When unconfiguring a queue, the *queue_handle* (*arg0*) represents the queue to be unconfigured.

The *page_size* (*arg1*) specifies the size of the page to be configured, encoded according to Table 21-18 of [ua2007n3] (that is 3 for 4M and 5 for 256M, other values are invalid).

The *queue_size* (*arg2*) specifies the number of entries in the queue and must be a power of 2. A value of zero (0) is used to unconfigure the given queue represented by *queue_handle* (*arg0*).

Note

The queue being configured is only of the CWQ for the core containing the CPU upon which the caller is executing. The calling thread should bind itself to the current CPU to ensure its context does not get switched to a different CPU and possibly a different core during the operation.

25.2.10.1. Errors

EINVAL	The specified *queue_size* is not a power of two, the *page_size* is invalid, or the *queue_handle* is invalid
ENOACCESS	CPU does not have access to a CWQ
EBADALIGN	*base_address* is improperly aligned
ENORADDR	*base_address* is not a valid real address

25.3. Trusted Platform Module Physical Access

A platform's TPM is expected to be used by a guest assuming the role of the control domain. These APIs are intended to be used by that guest. Virtual TPMs are expected to be used by all other guests.

25.3.1. TPM Definitions

25.3.1.1. TPM Locality

Locality is defined in the PC Client specification[tpmpc]. Following is a brief summary of TPM Localities.

Locality is a concept that allows various trusted processes on the platform to communicate with the TPM such that the TPM is aware of which trusted process is sending commands. There are six Localities defined (numbered 0-4 and Legacy). Their use is defined as:

Locality 4	Trusted hardware. This is the Dynamic RTM.
Locality 3	Auxiliary components. Use of this is optional and, if used, it is implementation dependent.
Locality 2	This is the run-time environment for the Trusted Operating System.
Locality 1	An environment for use by the Trusted Operating System.
Locality 0	The legacy environment for the Static RTM and its chain of trust.
Legacy locality	This is Locality 0 using TPM 1.1 type I/O ports.

25.3.1.2. TPM Registers

The registers specified by the PC Client specification[tpmpc] for Locality 0 are listed below.

Table 25.5. TPM Registers

Offset	Size	Register	Description
0x00	1	`TPM_ACCESS`	Used to gain ownership of port
0x08	4	`TPM_INT_ENABLE`	Controls interrupts
0x0c	1	`TPM_INT_VECTOR`	SIRQ vector to be used by the TPM
0x10	4	`TPM_INT_STATUS`	Interrupt status
0x14	4	`TPM_INTF_CAPABILITY`	Interrupt capabilities
0x18	4	`TPM_STS`	Status register
0x24	4	`TPM_DATA_FIFO`	Read or write FIFO
0xf00	4	`TPM_DID_VID`	Device ID/Vendor ID
0xf04	1	`TPM_RID`	Device revision ID
0xf05-0xf7f			TCG-defined config registers
0xf80-0xfff			Vendor-defined config registers

All registers are architected to be exposed to “untrusted” software. There are various protocols to disable certain functionality before the “untrusted” software is handed control. The guest may have access to all of the programmable registers just as in a typical personal computing environment.

25.3.2. TPM Hypervisor Calls

The following API calls are used to access a platform's TPM and are in the TPM API group.

25.3.2.1. tpm_get

trap#	`FAST_TRAP`
function#	`TPM_GET`
arg0	*`locality`*
arg1	*`register_offset`*

arg2	*access_size*
ret0	*status*
ret1	*register_value*

This call reads the value of a TPM register as specified by the *register_offset* argument using the TPM locality specified by *locality*. The size of the register access is specified by the *access_size* argument.

On success, the call returns a *status* of EOK and the value of the requested TPM register.

25.3.2.1.1. Errors

EINVAL	Invalid *locality*, *register_offset*, or *access_size*.
ENOACCESS	Operation administratively prohibited.

25.3.2.2. tpm_put

trap#	FAST_TRAP
function#	TPM_PUT
arg0	*locality*
arg1	*register_offset*
arg2	*access_size*
arg3	*register_value*
ret0	*status*

This call writes a value to a TPM register as specified by the *register_offset* argument using the TPM locality specified by *locality*. The size of the register access is specified by the *access_size* argument.

On success, the call returns a *status* of EOK and the value of the requested TPM register.

25.3.2.2.1. Errors

EINVAL	Invalid locality, register_offset, or access_size.
ENOACCESS	Operation administratively prohibited.

Chapter 26. UltraSPARC-T2 Network Interface Unit

26.1. Introduction

The network interface incorporated into the UltraSPARC-T2 processor is designed to be high performance and capable of many sophisticated operations in order to optimize the performance of the UltraSPARC strands themselves.

Critically in support of virtualization, the device has multiple DMA engines that can be assigned to different on-chip processing strands and driven by an on-chip packet filter in order to balance packet processing load and achieve the greatest possible parallelism.

A detailed discussion of this device is beyond the scope of this document, and the reader is recommended to read Chapters 22 through 28 [ua2007n2] for more detail.

For the purpose of this document, we assume a working knowledge of the NIU and concern ourselves with accessing the device via programmed IO operations (PIOs) and the addresses used in read/write requests. The latter relates to memory protection. Together these two features enable resource (both memory and DMAs) isolation, which is the basis of virtualization.

In UltraSPARC T2, since the device is part of the processor, the hypervisor controls how the hardware is presented to a guest OS. Not all hardware resources support virtualization directly.

The NIU in UltraSPARC T2 is accessed primarily via load and store instructions and the hypervisor may organize the hardware as a two-function device split into two different address ranges. Within each function, two address ranges are defined: one for management, one for virtualization. The entire device may be accessed through the management addresses. Virtualization addresses, on the other hand, only have accesses to a set of defined DMAs.

The control and status registers (CSRs) of multiple DMA channels can be grouped into an 8KB page within the virtualization address ranges. The grouping itself is defined by a table in the management address range. To support memory protection, each transmit or receive DMA supports two logical pages. The addresses in the configuration registers, packet gather list pointers on the transmit side, and the allocated buffer pointer on the receive side will be relocated accordingly. The logical page registers are only accessible via the management address ranges.

In UltraSPARC T2, the sun4v hypervisor software may expose an 8KB page, with a few DMAs defined, to the driver software thus enabling the driver software to control those DMAs via PIOs. In addition, hypervisor also defines the logical page registers for these DMAs, which limits the addresses ranges allowed in the descriptors for DMA transactions. Together, this protects the system memory with regard to DMA operations guest OS software may use.

The remainder of this section details the hypervisor APIs calls available to interact with the UltraSPARC-T2 NIU, however a working knowledge of the device is essential to understand these interfaces.

26.2. Definitions

Here we define a few of the abbreviations and acronyms used in the rest of this section.

Logical Device (LD) - A term used generically to refer to a functional block that may ultimately cause an interrupt.

Logical Device Group (LDG) - A group of logical devices sharing an interrupt. A group may have only one LD.

Logical Device Flag (LDF) - Is a logical 'OR' of some LC

Logical Device Group Interrupt (LDGI) - The interrupt associated with a LDG. This interrupt is controlled by a one shot mechanism, i.e. hardware will issue only one single interrupt, and software will need to arm the LDG again to enable it to issue another interrupt.

Logical Device State Vector (LDSV) - a read only state vector capturing the LDFs of ALL the LDs.

Logical Domain (LDom) - Separation of platform resources into self-contained partition that is capable of supporting an operating system.

Logical Page - A contiguous range of memory location. If an address posted by software is within the logical page, it will be translated to a physical address by replacing the base address of the logical page with the base address of the physical page. The size of the logical page is programmable.

Receive Block Ring (RBR) - It is a ring buffer of memory blocks posted by software.

Receive Completion Ring (RCR) - The ring stores the addresses of the buffers used to store incoming packets.

Receive DMA Channel (RDC) - It is comprised of a RBR, a RCR and a set of control and status registers. A receive DMA channel is selected after an incoming packet is classified. A packet buffer is derived from the pool and used to store the incoming packet. Each channel is capable of issuing interrupt to software based on the queue length of the Receive Completion Ring or a time-out.

Transmit Ring (TR) - The data structure built in system memory for software to post transmission requests.

Transmit DMA Channel (TDC) - Consists of a transmit ring and a set of control and status registers.

26.3. Version 1.0 and version 1.1 APIs

Version 1.0 of the NIU APIs allow a domain that owns a complete NIU device to configure, manage and send/receive data through the NIU device.

Version 1.1 of the NIU APIs extend this ability to allow a domain to own part of the NIU device, specifically a virtual region with associated resources. It also adds a set of APIs to enable the domain that owns the NIU device to share it with another domain.

26.4. Version 1.0 APIs

The following APIs are available by negotiating version 1.0 for the NIU API group.

26.4.1. niu_rx_logical_page_set

trap#	`FAST_TRAP`
function#	`N2NIU_RX_LP_SET`
arg0	*`chidx`*
arg1	*`pgidx`*
arg2	*`raddr`*
arg3	*`size`*
ret0	*`status`*

This API configures a mapping described by arguments raddr and size in the NIU receive DMA engine address translation (logical page) register indicated by chidx and pgidx.

If there is already a valid mapping for the page specified by pgidx, that mapping is overwritten.

The specified mapping is unconfigured if the size is 0. In this case, raddr is ignored.

chidx must be between 0 and 15.

pgidx must be 0 or 1.

raddr must be size aligned.

size must be a power of 2.

26.4.1.1. Errors

EBADALIGN	Invalid alignment for raddr or size
ENORADDR	Invalid real address
EINVAL	Invalid index for channel or register

26.4.2. niu_rx_logical_page_get

trap#	`FAST_TRAP`
function#	`N2NIU_RX_LP_GET`
arg0	*`chidx`*
arg1	*`pgidx`*
ret0	*`status`*
ret1	*`raddr`*
ret2	*`size`*

Return the current mapping in the NIU receive DMA engine address translation (logical page) register indicated by *`chidx`* and *`pgidx`*. The real address and size are returned in *`ret1`* and *`ret2`*.

`chidx` must be between 0 and 15.

`pgidx` must be 0 or 1.

If there is no current mapping for the given *`chidx`* and *`pgidx`*, then the return values *`raddr`* and *`size`* will both be 0.

26.4.2.1. Errors

EINVAL	Invalid index for channel or register

26.4.3. niu_tx_logical_page_set

trap#	`FAST_TRAP`
function#	`N2NIU_TX_LP_SET`
arg0	*`chidx`*
arg1	*`pgidx`*
arg2	*`raddr`*

arg3	*size*
ret0	*status*

Configure a mapping described by arguments *raddr* and *size* in the NIU transmit DMA engine address translation (logical page) register indicated by *chidx* and *pgidx*.

If there is already a valid mapping for the page specified by *pgidx*, that mapping is overwritten. The specified mapping is unconfigured if the size is 0. In this case, *raddr* is ignored.

chidx must be between 0 and 15.

pgidx must be 0 or 1.

raddr must be size-aligned.

size must be a power of 2.

26.4.3.1. Errors

EBADALIGN	Invalid alignment for *raddr* or *size*
ENORADDR	Invalid real address
EINVAL	Invalid index for channel or register

26.4.4. niu_tx_logical_page_get

trap#	FAST_TRAP
function#	N2NIU_TX_LP_GET
arg0	*chidx*
arg1	*pgidx*
ret0	*status*
ret1	*raddr*
ret2	*size*

Return the current mapping in the NIU transmit DMA engine address translation (logical page) register indicated by *chidx* and *pgidx*. The real address and size are returned in *ret1* and *ret2*.

chidx must be between 0 and 15.

pgidx must be 0 or 1.

If there is no current mapping for the given *chidx* and *pgidx*, then the return values *raddr* and *size* will both be 0.

26.4.4.1. Errors

EINVAL	Invalid index for channel or register

26.5. Version 1.1 APIs

Version 1.1 APIs are an extension to the preceding version 1.0 APIs. The preceding APIs continue to function, however by successfully negotiating version 1.1 for the NIU API group the following APIs will also be available for guest software running on a UltraSPARC-T2 system.

26.5.1. NIU Virtual Region (VR) Specific APIs

26.5.1.1. vr_assign

trap#	FAST_TRAP
function#	N2NIU_VR_ASSIGN
arg0	*vr_idx*
arg1	*ldc_id*
ret0	*status*
ret1	*vr_cookie*

This API assigns the specified virtual region to a domain identified by the endpoint *ldc_id* of the channel to the target domain. The returned *vr_cookie* can be used by a domain to obtain access to the virtual region.

vr_idx is the Virtualization Region index number (0–7). The NIU has 2 Functions, each Function has 2 Virtualization regions, each region can be split into 2 access protected pages.

The *ldc_id* is the LDC endpoint in the domain that owns the NIU device and the channel that runs between and the domain to which the virtual region is being assigned.

Upon success the API returns in *vr_cookie* a 32-bit unique id. This cookie represents a specific NIU and a specific Virtual Region (VR) within it.

26.5.1.1.1. Errors

ENOACCESS	Domain does not own the NIU
ECHANNEL	Invalid channel (LDC ID)
EINVAL	Invalid *vr_idx* or VR is already assigned

26.5.1.2. vr_unassign

trap#	FAST_TRAP
function#	N2NIU_VR_UNASSIGN
arg0	*vr_cookie*
ret0	*status*

This API frees the virtual region that was previously assigned to a domain. Only the domain that owns the NIU device is allowed to call this interface. After the virtual region is unassigned, subsequent access by the guest will fail with EINVAL to hypervisor calls, or memory access violations.

vr_cookie is a 32-bit unique id that represents the NIU virtual region as returned by n2niu_vr_assign.

26.5.1.2.1. Errors

ENOACCESS	Domain does not own the NIU
EINVAL	Invalid cookie or VR not assigned

26.5.1.3. vr_getinfo

trap#	FAST_TRAP
function#	N2NIU_VR_GETINFO
arg0	*vr_cookie*
ret0	*status*
ret1	*real_base*
ret2	*real_size*

This API obtains the real address base and size for the virtual region corresponding to the specified cookie value. This API can only successfully be called from the guest that owns the virtual region associated with that cookie.

vr_cookie A 32-bit unique id that represents a NIU/VR.

real_base is the base real address of the start of the virtualization region. *real_size* is the size of the VR mapping.

26.5.1.3.1. Errors

ENOACCESS	Cookie not associated with this domain
EINVAL	Invalid cookie

26.5.2. NIU DMA Channel (DMAC) Specific APIs

26.5.2.1. vr_rx_dma_assign and vr_tx_dma_assign

trap#	FAST_TRAP
function#	N2NIU_VR_RX_DMA_ASSIGN
arg0	*vr_cookie*
arg1	*gch_idx*
ret0	*status*
ret1	*vch_idx*

trap#	FAST_TRAP
function#	N2NIU_VR_TX_DMA_ASSIGN
arg0	*vr_cookie*
arg1	*gch_idx*
ret0	*status*
ret1	*vch_idx*

These two APIs assign TX and RX DMA channel resources to a specific virtual region. A virtual region has to be assigned to a domain before resources can be assigned to the virtual region. There is a hardware maximum of 8 channels per virtual region, but implementations may restrict the channels maximum further. Each global channel may only be assigned to one virtual region at a time.

vr_cookie is a 32-bit unique id that represents an NIU/VR.

gch_idx is the Global DMA channel index number (0–15).

vch_idx is the Virtual DMA channel index number (0–7).

Programming Note

The interrupt resources assigned to this channel will be automatically migrated to the guest domain. In addition, the interrupt resource is also marked disabled. Its the responsibility of the domain that owns the NIU device to remove any interrupt handler associated with the channel.

26.5.2.1.1. Errors

ENOACCESS	Domain does not own the NIU
EINVAL	Invalid cookie or channel
ENOMAP	Channel not available

26.5.2.2. vr_rx_dma_unassign and vr_tx_dma_unassign

trap#	FAST_TRAP
function#	N2NIU_VR_RX_DMA_UNASSIGN
arg0	*vr_cookie*
arg1	*vch_idx*
ret0	*status*

trap#	FAST_TRAP
function#	N2NIU_VR_TX_DMA_UNASSIGN
arg0	*vr_cookie*
arg1	*vch_idx*
ret0	*status*

This API unassigns RX and TX DMA channel resources from a virtual region. Accesses to an unassigned virtual channel in the guest will return EINVAL or memory access violations.

Once a channel has been unassigned it may be reassigned to another region.

vr_cookie is a 32-bit unique id that represents the virtual region.

vch_idx is the Virtual DMA channel index number (0–7).

Programming Note

The unassign operation will migrate the interrupts back to the domain that owns the NIU device. It will also disable the channel if it is not already disabled. The channels are restored back to the domain that owns the NIU device.

26.5.2.2.1. Errors

ENOACCESS	Domain does not own the NIU
EINVAL	Invalid cookie or channel

ENOMAP	Channel is not assigned

26.5.2.3. vr_get_rx_map and vr_get_tx_map

trap#	FAST_TRAP
function#	N2NIU_VR_GET_RX_MAP
arg0	*vr_cookie*
ret0	*status*
ret1	*dma_map*

trap#	FAST_TRAP
function#	N2NIU_VR_GET_TX_MAP
arg0	*vr_cookie*
ret0	*status*
ret1	*dma_map*

These APIs obtain a list of TX or RX DMA channel resources assigned to a virtual region.

vr_cookie is a 32-bit unique id that represents an NIU/VR. Upon success the API returns in *dma_map* the Rx/Tx DMA channel map (bit mask) that shows which slots in the virtual region have DMA channels mapped. For example, bit *N* will be set in the map iff virtual channel *N* (0–7) is assigned in the VR.

26.5.2.3.1. Errors

ENOACCESS	Cookie not associated with this domain
EINVAL	Invalid cookie

26.5.2.4. vrrx_set_ino and vrtx_set_ino

trap#	FAST_TRAP
function#	N2NIU_VRRX_SET_INO
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*ino*
ret0	*status*

trap#	FAST_TRAP
function#	N2NIU_VRTX_SET_INO
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*ino*
ret0	*status*

This API assigns an interrupt number for the specified RX/TX virtual DMA channel in a virtual region. A unique interrupt number should be assigned to each channel across all VRs assigned from a single NIU device.

vr_cookie is a 32-bit unique id that represents an NIU/VR. *vch_idx* is the Virtual DMA channel index number, retrieved through the n2niu_vr_get_*_map interface (0–7). *ino* is a unique 32-bit device interrupt no. (devino) to be associated with this channel. Each DMA Channel corresponds to an interrupt source and should be assigned a unique interrupt number between 0 and 63.

Programming Note

These device inos must then be assigned an interrupt cookie, (or converted to system wide interrupt numbers sysinos), for use within the domain.

26.5.2.4.1. Errors

ENOACCESS	Cookie not associated with this domain
EINVAL	Invalid cookie

26.5.2.5. vrrx_get_info and vrtx_get_info

trap#	FAST_TRAP
function#	N2NIU_VRTX_SET_INO
arg0	*vr_cookie*
arg1	*vch_idx*
ret0	*status*
ret1	*group*
ret2	*logdev*

trap#	FAST_TRAP
function#	N2NIU_VRTX_SET_INO
arg0	*vr_cookie*
arg1	*vch_idx*
ret0	*status*
ret1	*group*
ret2	*logdev*

These APIs get the virtual group number and logical device associated with a RX/TX virtual DMA channel in a virtual region. Since interrupts are delivered via bits in the LDSV that corresponds to the logical device, the guest needs to map each virtual channel to a logical device in order to identify the interrupted channel and re-arm the interrupt. The guest will use PIO's using these values to rearm the associated interrupts.

vr_cookie is a 32-bit unique id that represents an NIU/VR.

vch_idx is the Virtual DMA channel index number (0–7).

Upon success the API returns in *group* the Virtual Group number (Bits 7:5 of the VRARDDR associated with that VR's LDSV management, and in *logdev* the logical device number. Please refer to [ua2007n2] for more detail.

26.5.2.5.1. Errors

ENOACCESS	Cookie not associated with this domain

EINVAL	Invalid cookie
ENOINTR	No virtual group exists for that channel in this domain

26.5.2.6. vrrx_lp_set and vrtx_lp_set

trap#	FAST_TRAP
function#	N2NIU_VRRX_LP_SET
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*pgidx*
arg3	*raddr*
arg4	*size*
ret0	*status*

trap#	FAST_TRAP
function#	N2NIU_VRTX_LP_SET
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*pgdix*
arg2	*raddr*
arg2	*size*
ret0	*status*

These APIs configure a mapping described by arguments raddr and size in the NIU DMA engine address translation (logical page) register indicated by vch_idx and pgidx. If there is already a valid mapping for the page specified by pgidx, that mapping is overwritten. The specified mapping is unconfigured if the size is 0. In this case, raddr is ignored. If the size is non-zero, the real address (raddr) should be size aligned and the size must be a power of 2.

This interface is identical to the version 1.0 NIU interfaces described above except for the presence of a cookie, and it uses virtual channels instead of global channels. Accessing this memory after the region has been unassigned will cause access violations in the guest.

The argument vr_cookie is a 32-bit unique id that represents an NIU/VR. vch_idx is the virtual DMA channel index number and should be between 0 and 15. pgidx is the logical page index number and legal values are 0 or 1. raddr is the logical page real address (size aligned) and size is the logical page size.

26.5.2.6.1. Errors

ENOACCESS	Cookie not associated with this domain
EBADALIGN	Invalid alignment for raddr or size
EINVAL	Invalid cookie or invalid index for channel or page

26.5.2.7. vrrx_lp_get and vrtx_lp_get

trap#	FAST_TRAP
function#	N2NIU_VRRX_LP_GET

arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*pgidx*
ret0	*status*
ret1	*raddr*
ret2	*size*

trap#	FAST_TRAP
function#	N2NIU_VRTX_LP_GET
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*pgidx*
ret0	*status*
ret1	*raddr*
ret2	*size*

These APIs return the current mapping in the NIU DMA engine address translation (logical page) register indicated by vch_idx and pgidx. The real address and size are returned to the caller. If there is no current mapping for the given chidx and pgidx, then the return values raddr and size will both be 0. This interface is identical to the NIU version 1.0 interfaces except for the presence of a cookie, and it uses virtual channels instead of global channels.

The argument vr_cookie is a 32-bit unique id that represents an NIU/VR. vch_idx is the virtual DMA channel index number and should be in the range 0 to 7. pgidx is the logical page index number; legal values are 0 and 1.

The APIs return raddr the logical page real address and size the logical page size.

26.5.2.7.1. Errors

ENOACCESS	Cookie not associated with this domain
EINVAL	Invalid cookie or invalid index for channel or page

26.5.3. Virtualized Access to Non-virtualized NIU registers

The domain that is the recipient of a virtual region and its DMA channel resources is only allowed limited access to various registers that control DMA behavior. The APIs specified below allow the domain to set or get non-virtualized DMA channel registers.

26.5.3.1. vrrx_param_get and vrtx_param_get

trap#	FAST_TRAP
function#	N2NIU_VRRX_PARAM_GET
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*param*
ret0	*status*

ret1	*value*

trap#	FAST_TRAP
function#	N2NIU_VRTX_PARAM_GET
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*param*
ret0	*status*
ret1	*value*

These APIs return the current value of a RX/TX virtual channel parameter. Where vr_cookie is a 32 bit unique id that represents an NIU/VR. vch_idx is the Virtual DMA channel index number, and param is the register to query (enumerated lookup) Upon success value contains the register value.

Legal Values for RX params (others return EINVAL):

Register	Value	Reference
RDC_RED_PARA	0	[ua2007n2], Table 25-19

Legal Values for TX params (others return EINVAL):

Register	Value	Reference
TDC_DMA_MAX	0	[ua2007n2], Table 26-25

26.5.3.1.1. Errors

ENOACCESS	Cookie not associated with this domain or specified parameter is not accessible
EINVAL	Invalid cookie or invalid index for channel or page

26.5.3.2. vrrx_param_set and vrtx_param_set

trap#	FAST_TRAP
function#	N2NIU_VRRX_PARAM_SET
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*param*
arg3	*value*
ret0	*status*

trap#	FAST_TRAP
function#	N2NIU_VRTX_SET_INO
arg0	*vr_cookie*
arg1	*vch_idx*
arg2	*param*
arg3	*value*
ret0	*status*

These APIs set the value of a RX/TX virtual channel parameter. Where *vr_cookie* is a 32-bit unique id that represents an NIU/VR. *vch_idx* is the Virtual DMA channel index number. *param* specifies the register to set and *value* is the register value.

Legal Values for RX params (others return EINVAL):

Register	Value	Reference
RDC_RED_PARA	0	[ua2007n2], Table 25-19

Legal Values for TX params (others return EINVAL):

Register	Value	Reference
TDC_DMA_MAX	0	[ua2007n2], Table 26-25

26.5.3.2.1. Errors

ENOACCESS	Cookie not associated with this domain or specified parameter is not accessible
EINVAL	Invalid cookie or invalid index for channel or page

26.6. Version 2.0 APIs

The current set of NIU APIs allow a domain that owns the device to configure, manage and send/receive data through the NIU device, or to assign HW resources to another logical domain.

The new set of hypervisor APIs proposed in this document extend these APIs to reference a specific NIU on the system. The NIU will be identified using a devhandle — a unique identifier that can be obtained by reading the cfg-handle property specified in the NIU device node in the guest MD.

The following APIs are available by negotiating version 2.0 for the NIU API group:

Modified	N2NIU_RX_LP_SET
	N2NIU_RX_LP_GET
	N2NIU_TX_LP_SET
	N2NIU_TX_LP_GET
	N2NIU_VR_ASSIGN
Unmodified	N2NIU_VR_UNASSIGN
	N2NIU_VR_GETINFO
	N2NIU_VR_RX_DMA_ASSIGN
	N2NIU_VR_RX_DMA_UNASSIGN
	N2NIU_VR_TX_DMA_ASSIGN
	N2NIU_VR_TX_DMA_UNASSIGN
	N2NIU_VR_GET_RX_MAP
	N2NIU_VR_GET_TX_MAP
	N2NIU_VRRX_SET_INO
	N2NIU_VRTX_SET_INO
	N2NIU_VRRX_GET_INFO
	N2NIU_VRTX_GET_INFO

N2NIU_VRRX_LP_SET
N2NIU_VRRX_LP_GET
N2NIU_VRTX_LP_SET
N2NIU_VRTX_LP_GET
N2NIU_VRRX_PARAM_GET
N2NIU_VRRX_PARAM_SET
N2NIU_VRTX_PARAM_GET
N2NIU_VRTX_PARAM_SET

26.6.1. niu_rx/tx_logical_page_set

trap#	FAST_TRAP
function#	N2NIU_RX_LP_SET
arg0	*devhandle*
arg1	*chidx*
arg2	*pgidx*
arg3	*raddr*
arg4	*size*
ret0	*status*

trap#	FAST_TRAP
function#	N2NIU_TX_LP_SET
arg0	*devhandle*
arg1	*chidx*
arg2	*pgidx*
arg3	*raddr*
arg4	*size*
ret0	*status*

These interfaces are identical to Section 26.4.1, “niu_rx_logical_page_set” and Section 26.4.3, “niu_tx_logical_page_set” except for the addition of the *devhandle* argument.

26.6.2. niu_rx/tx_logical_page_get

trap#	FAST_TRAP
function#	N2NIU_RX_LP_GET
arg0	*devhandle*
arg1	*chidx*
arg2	*pgidx*
ret0	*status*
ret1	*raddr*
ret2	*size*

trap#	FAST_TRAP

function#	N2NIU_TX_LP_GET
arg0	*devhandle*
arg1	*chidx*
arg2	*pgidx*
ret0	*status*
ret1	*raddr*
ret2	*size*

These interfaces are identical to Section 26.4.2, “niu_rx_logical_page_get” and Section 26.4.4, “niu_tx_logical_page_get” except for the addition of the *devhandle* argument.

26.6.3. NIU Virtual Region (VR) Specific APIs

26.6.3.1. vr_assign

trap#	FAST_TRAP
function#	N2NIU_VR_ASSIGN
arg0	*devhandle*
arg1	*vr_idx*
arg2	*ldc_id*
ret0	*status*
ret1	*vr_cookie*

This interface is identical to Section 26.5.1.1, “vr_assign” except for the addition of the *devhandle* argument.

Chapter 27. Chip and platform specific performance counters

27.1. UltraSPARC-T1 performance counters

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.

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

Table 27.1. UltraSPARC-T1 J-Bus/DRAM Performance Counters

PerfReg	Description
0	J-Bus Performance control register
1	J-Bus 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

27.1.1. 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.

27.1.1.1. Errors

ENOACCESS	No access allowed to performance registers

EINVAL	Invalid performance register number

27.1.2. 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 *perfreg*, to value. Upon successful completion, it updates the specified performance register value and returns EOK status.

27.1.2.1. Errors:

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

27.2. UltraSPARC-T1 MMU statistics counters

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

27.2.1. Hypervisor API for UltraSPARC-T1 MMU statistics collection

On UltraSPARC-T1, 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).

27.2.1.1. MMU statistic buffer layout

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

Table 27.2. UltraSPARC-T1 MMU statistic buffer layout

Offset	Size	Field
0x00	0x08	IMMU TSB hits ctx0, 8KByte TTE
0x08	0x08	IMMU TSB ticks ctx0, 8KByte TTE
0x10	0x08	IMMU TSB hits ctx0, 64KByte TTE
0x18	0x08	IMMU TSB ticks ctx0, 64KByte TTE
0x20	0x10	reserved
0x30	0x08	IMMU TSB hits ctx0, 4MByte TTE
0x38	0x08	IMMU TSB ticks ctx0, 4MByte TTE
0x40	0x10	reserved
0x50	0x08	IMMU TSB hits ctx0, 256MByte TTE
0x58	0x08	IMMU TSB ticks ctx0, 256MByte TTE
0x60	0x20	reserved
0x80	0x08	IMMU TSB hits ctxnon0, 8KByte TTE

Offset	Size	Field
0x88	0x08	IMMU TSB ticks ctxnon0, 8KByte TTE
0x90	0x08	IMMU TSB hits ctxnon0, 64KByte TTE
0x98	0x08	IMMU TSB ticks ctxnon0, 64KByte TTE
0xa0	0x10	reserved
0xb0	0x08	IMMU TSB hits ctxnon0, 4MByte TTE
0xb8	0x08	IMMU TSB ticks ctxnon0, 4MByte TTE
0xc0	0x10	reserved
0xd0	0x08	IMMU TSB hits ctx0, 256MByte TTE
0xd8	0x08	IMMU TSB ticks ctx0, 256MByte TTE
0xe0	0x20	reserved
0x100	0x08	DMMU TSB hits ctx0, 8KByte TTE
0x108	0x08	DMMU TSB ticks ctx0, 8KByte TTE
0x110	0x08	DMMU TSB hits ctx0, 64KByte TTE
0x118	0x08	DMMU TSB ticks ctx0, 64KByte TTE
0x120	0x10	reserved
0x130	0x08	DMMU TSB hits ctx0, 4MByte TTE
0x138	0x08	DMMU TSB ticks ctx0, 4MByte TTE
0x140	0x10	reserved
0x150	0x08	DMMU TSB hits ctx0, 256MByte TTE
0x158	0x08	DMMU TSB ticks ctx0, 256MByte TTE
0x160	0x20	reserved
0x180	0x08	DMMU TSB hits ctxnon0, 8KByte TTE
0x188	0x08	DMMU TSB ticks ctxnon0, 8KByte TTE
0x190	0x08	DMMU TSB hits ctxnon0, 64KByte TTE
0x198	0x08	DMMU TSB ticks ctxnon0, 64KByte TTE
0x1a0	0x10	reserved
0x1b0	0x08	DMMU TSB hits ctxnon0, 4MByte TTE
0x1b8	0x08	DMMU TSB ticks ctxnon0, 4MByte TTE
0x1c0	0x10	reserved
0x1d0	0x08	DMMU TSB hits ctx0, 256MByte TTE
0x1d8	0x08	DMMU TSB ticks ctx0, 256MByte TTE
0x1e0	0x20	reserved

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

27.2.2. 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.

27.2.2.1. Errors

ENORADDR	Invalid raddr
EBADALIGN	raddr not aligned on a 64-byte boundary
EBADTRAP	API not supported (all non-UltraSPARC-T1 architectures)

27.2.3. 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`*.

27.2.3.1. Errors

EBADTRAP	API not supported (all non-UltraSPARC-T1 architectures)

27.3. Fire performance counter APIs

The UltraSPARC-T1 processor is connected to its IO sub-systems via Sun's J-Bus interconnect. The Fire I/O ASIC is used in most UltraSPARC-T1 based systems to bridge between this J-Bus and two PCI-Express root complexes. The SPARC Hypervisor virtualizes and mostly hides this physical infrastructure. This set of APIs, when available, provide limited access to the internal performance counters of the Fire device.

27.3.1. Definitions

For the purpose of accessing Fire performance counters devhandle as defined in Section 23.2, “IO Data Definitions” is used to identify the Fire bridge, (and consequently its performance counters), associated with a particular PCI-Express root complex.

Within each Fire each performance register is assigned a unique performance register (PerfReg) number for reading/writing purposes as follows:

Table 27.3. Fire performance counters

PerfReg	Description
0	J-Bus Performance control register
1	J-Bus Performance counter register 0
2	J-Bus Performance counter register 1
3	PCIe IMU Performance control register
4	PCIe IMU Performance counter register 0
5	PCIe IMU Performance counter register 1
6	PCIe MMU Performance control register
7	PCIe MMU Performance counter register 0
8	PCIe MMU Performance counter register 1
9	PCIe TLU Performance control register
10	PCIe TLU Performance counter register 0
11	PCIe TLU Performance counter register 1
12	PCIe TLU Performance counter register 2
13	PCIe LPU Performance control register
14	PCIe LPU Performance counter register 0
15	PCIe LPU Performance counter register 1

The values associated with each performance counter are defined in the Fire 2.0 Programmer's Reference Manual, however performance register IDs 14 and 15 are implemented as read/write instead of read only.

27.3.2. fire_get_perf_reg

trap#	FAST_TRAP
function#	FIRE_GET_PERFREG
arg0	devhandle
arg1	perfreg
ret0	status
ret1	value

This call reads the value of the Fire performance register specified by the argument perfreg of the Fire leaf specified by the argument devhandle.

Upon successful completion, it returns EOK status and performance register value. Otherwise, it returns one of the following errors:

27.3.2.1. Errors

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

27.3.3. fire_set_perf_reg

trap#	FAST_TRAP

function#	FIRE_SET_PERFREG
arg0	devhandle
arg1	perfreg
arg2	value
ret0	status

This call sets the value of the Fire performance register as specified by the argument perfreg of the Fire leaf specified by the argument devhandle to the value specified by the argument value.

Upon successful completion, it updates the specified performance register value and returns EOK status. Otherwise, it returns one of the following errors:

27.3.3.1. Errors

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

27.4. UltraSPARC T2 performance counters

The UltraSPARC-T2 processor is a fully integrated System On a Chip (SOC) design that incorporates processing cores together with memory controllers, a PCI Express IO root complex and high performance Ethernet interfaces. Performance instrumentation is provided on-chip for each of SPARC, DRAM, PCI-Express and Ethernet sub-systems.

27.4.1. Strand performance instrumentation

Each hardware strand has a pair of registers to control/capture CPU specific instrumentation:

Table 27.4. SPARC performance counters

Description	Access
SPARC Performance Control Register	ASR 0x10
SPARC Performance Instrumentation Counter	ASR 0x11

These registers are directly accessible by the privileged code. The HT bit in SPARC PCR controls the counting of hyperprivileged events, can be set only in hyperprivileged mode. The hypervisor provides an API to allow read/write access to the SPARC performance control register. A guest should not assume it can count hyperprivileged events. Attempting to set HT bit may result in the API call failing with ENOACCESS and the guest should handle this gracefully.

For further information on the register specifications the reader is directed to the UltraSPARC-T2 programmers reference manual.

27.4.2. DRAM Performance Instrumentation

Each DRAM channel in Niagara2 has a pair of performance counters, packed into a single register, plus a register to control what is counted. There are a total of four different DRAM channels for a UltraSPARC-T2 system. The hypervisor provides an API for read/write access to these registers.

27.4.3. API calls for SPARC and DRAM performance counters

Each of the SPARC and DRAM controller performance registers is assigned a unique performance register (PerfReg) number as follows:

Table 27.5. UltraSPARC-T2 SPARC and DRAM performance counters

PerfReg	Description
0	SPARC Performance Control Register
1	DRAM Performance Control Register 0
2	DRAM Performance Counter Register 0
3	DRAM Performance Control Register 1
4	DRAM Performance Counter Register 1
5	DRAM Performance Control Register 2
6	DRAM Performance Counter Register 2
7	DRAM Performance Control Register 3
8	DRAM Performance Counter Register 3

The interface for reading/writing SPARC performance control register will pass the entire register value and not just the HT bit.

27.4.4. niagara2_get_perfreg

trap#	`FAST_TRAP`
function#	`NIAGARA2_GET_PERFREG`
arg0	perfreg
ret0	status
ret1	value

This call reads the value of the SPARC or DRAM performance register, as specified by the argument perfreg.

Upon successful completion the call returns a status of EOK and a performance register value.

27.4.4.1. Errors

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

27.4.5. niagara2_set_perfreg

trap#	`FAST_TRAP`
function#	`NIAGARA2_GET_PERFREG`
arg0	perfreg
arg1	value
ret0	status

This calls sets the SPARC / DRAM performance register specified by the argument perfreg, to the value specified by the argument value.

Upon successful completion, it updates the specified performance register value and returns a status of EOK.

27.4.5.1. Errors

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

27.4.6. API calls for PCI-Express interface unit performance counters

The following hypervisor API calls provide access to the PCI Express Interface performance counters for a UltraSPARC-T2 processor.

The definition and functionality of the following performance registers is given in the UltraSPARC-T2 Programmer's Reference Manual.

Table 27.6. UltraSPARC-T2 PCI-Express performance counters

PerfReg	Description
0	DMU IMU Performance Counter Select
1	DMU IMU Performance Counter 0
2	DMU IMU Performance Counter 1
3	DMU MMU Performance Counter Select
4	DMU MMU Performance Counter 0
5	DMU MMU Performance Counter 1
6	PEU Performance Counter Select
7	PEU Performance Counter 0
8	PEU Performance Counter 1
9	PEU Performance Counter 2
10	PEU Bit Error Counter I
11	PEU Bit Error Counter II

27.4.7. n2piu_get_perf_reg

trap#	`FAST_TRAP`
function#	`N2PIU_GET_PERFREG`
arg0	devhandle
arg1	perfreg
ret0	status
ret1	value

This call reads the value of the UltraSPARC-T2 PIU performance register specified by the argument *`perfreg`* of the PCI leaf specified by the argument *`devhandle`*.

Upon successful completion, it returns EOK *`status`* and performance register *`value`*.

27.4.7.1. Errors

EINVAL	Invalid performance register number

ENOACCESS	No access allowed to performance registers

27.4.8. n2piu_set_perf_reg

trap#	FAST_TRAP
function#	N2PIU_SET_PERFREG
arg0	devhandle
arg1	perfreg
arg2	value
ret0	status

This call sets the value of the N2 PIU performance register as specified by the argument *perfreg* of the PCI leaf specified by the *devhandle* argument to the value specified by the argument *value*.

Upon successful completion, it updates the specified performance register value and returns EOK *status*.

27.4.8.1. Errors

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

27.5. UltraSPARC T2+ performance counters

The UltraSPARC-T2+ processor is a fully integrated System On a Chip (SOC) design that incorporates processing cores together with memory controllers, a PCI Express IO root complex, and coherency links for multi-node support. Performance instrumentation is provided on-chip for each of SPARC, DRAM, PCI-Express, and Coherency Link sub-systems.

27.5.1. Strand performance instrumentation

Each hardware strand has a pair of registers to control/capture CPU specific instrumentation:

Table 27.7. SPARC performance counters

Description	Access
SPARC Performance Control Register	ASR 0x10
SPARC Performance Instrumentation Counter	ASR 0x11

These registers are directly accessible by the privileged code. The HT bit in SPARC PCR controls the counting of hyperprivileged events, can be set only in hyperprivileged mode.

For further information on the register specifications the reader is directed to the UltraSPARC-T2+ programmers reference manual.

27.5.2. DRAM Performance Instrumentation

Each DRAM channel in UltraSPARC-T2+ has a pair of performance counters, packed into a single register, plus a register to control what is counted. There are a total of two DRAM channels on each node of an UltraSPARC-T2+ system. The hypervisor provides an API for read/write access to these registers.

27.5.3. L2 Cache Control Register

The L2 Control Register adds address interleave ceiling mask and nodeid fields. The PERF_CONFIG bits are the only bits that are exposed through these interfaces. The perf control bits in the L2 Control Register are 37:36 which are bits 1:0 in the virtualized L2 Control Register.

27.5.4. LPU Performance Instrumentation

There are three performance counter registers contained in the LPU core on a per port basis.

27.5.5. GPD Performance Instrumentation

There is one set of performance counter registers contained in the GPD core.

27.5.6. ASU Performance Instrumentation

There is one set of performance counter registers contained in the ASU core.

27.5.7. API calls for SPARC and DRAM performance counters

These sun4v APIs provide an interface to read and write the DRAM performance registers as they are not accessible by the privileged software.

The privileged software can use this interface to write the HT bit in the SPARC performance control register as well. The access to write to the HT bit can be denied, in which case the sun4v API returns ENOACCESS. The code using these interfaces must handle such a failure gracefully.

The SPARC performance registers are used to get CLC performance counters by setting the PERF_CONFIG bits in L2_CONTROL_REG. In default mode all L2 misses are counted. The PERF_CONFIG bits in L2 control register can be programmed to count misses serviced from local memory, misses serviced from remote memory or misses serviced by cache-to-cache transfers. As most of the L2_CONTROL_REG is not accessible by the privilege code, a virtualized generic register is used to program PERF_CONFIG bits in all L2_CONTROL_REG.

The SPARC PCR, the virtualized L2_CONTROL_REG, the DRAM, and Zambezi performance registers are assigned unique performance register numbers (PerfReg#) which uniquely identifies each performance register.

The API version 1.0 includes only the Victoria Falls SPARC and DRAM performance counters.

The API version 1.1 is an extension of version 1.0 and includes Zambezi performance counters as well.

The table below describes the SPARC PCR, the virtualized L2_CONTROL_REG, and the DRAM performance registers.

Table 27.8. UltraSPARC-T2+ SPARC, L2, and DRAM performance counters

PerfReg	Node	Group	Description
0	local		SPARC Performance Control Register
1	local	all	L2 Bank Control Register for local node
2	0	0	DRAM Performance Control Register 0
3	0	0	DRAM Performance Counter Register 0
4	0	1	DRAM Performance Control Register 0

PerfReg	Node	Group	Description
5	0	1	DRAM Performance Counter Register 0
6	1	0	DRAM Performance Control Register 0
7	1	0	DRAM Performance Counter Register 0
8	1	1	DRAM Performance Control Register 0
9	1	1	DRAM Performance Counter Register 0
10	2	0	DRAM Performance Control Register 0
11	2	0	DRAM Performance Counter Register 0
12	2	1	DRAM Performance Control Register 0
13	1	1	DRAM Performance Counter Register 0
14	3	0	DRAM Performance Control Register 0
15	3	0	DRAM Performance Counter Register 0
16	3	1	DRAM Performance Control Register 0
17	3	1	DRAM Performance Counter Register 0
18	0	LPU0	Zambezi 0 LPU A Performance Control Register
19	0	LPU0	Zambezi 0 LPU A Performance Counter Register 0
20	0	LPU0	Zambezi 0 LPU A Performance Counter Register 1
21	0	LPU1	Zambezi 0 LPU B Performance Control Register
22	0	LPU1	Zambezi 0 LPU B Performance Counter Register 0
23	0	LPU1	Zambezi 0 LPU B Performance Counter Register 1
24	0	LPU2	Zambezi 0 LPU C Performance Control Register
25	0	LPU2	Zambezi 0 LPU C Performance Counter Register 0
26	0	LPU2	Zambezi 0 LPU C Performance Counter Register 1
27	0	LPU3	Zambezi 0 LPU D Performance Control Register
28	0	LPU3	Zambezi 0 LPU D Performance Counter Register 0
29	0	LPU3	Zambezi 0 LPU D Performance Counter Register 1
30	0	GPD	Zambezi 0 GPD Performance Control Register
31	0	GPD	Zambezi 0 GPD Performance Counter Register 0
32	0	GPD	Zambezi 0 GPD Performance Counter Register 1
33	0	ASU	Zambezi 0 ASU Performance Control Register
34	0	ASU	Zambezi 0 ASU Performance Counter Register 0
35	0	ASU	Zambezi 0 ASU Performance Counter Register 1
36-53	1		Same as 18-35 but for Node 1
54-71	2		Same as 18-35 but for Node 2
72-89	3		Same as 18-35 but for Node 3

The interface for accessing the SPARC Performance Control Register operates on the entire register and always on the current hardware strand.

For Zambezi performance counters there is a case where multiple accesses cannot be satisfied given the hardware restrictions in Zambezi and where the hypervisor cannot block waiting to access Zambezi for

which API calls returns EWOULDBLOCK error code. A error code EINVAL is returned when the register number is out of range, i.e PerfReg# is not in 0-89. A error code ENOTSUPPORTED is returned when the PerfReg# is not supported, for e.g., Zambezi registers are not supported on a 2-way system. When a guest does not have access to a register the API call returns ENOACCESS error code.

27.5.8. vfalls_get_perfreg

trap#	FAST_TRAP
function#	VFALLS_GET_PERFREG
arg0	perfreg
ret0	status
ret1	value

This call reads the value of the SPARC, virtualized L2_CONTROL_REG, DRAM, or Zambezi performance register as specified by the argument *perfreg*.

Upon successful completion the call returns a *status* of EOK and a performance register *value*.

27.5.8.1. Errors

EINVAL	Invalid performance register number
ENOTSUPPORTED	Register number not supported
ENOACCESS	No access allowed to performance registers
EWOULDBLOCK	Cannot complete operation without blocking

27.5.9. vfalls_set_perfreg

trap#	FAST_TRAP
function#	VFALLS_GET_PERFREG
arg0	perfreg
arg1	value
ret0	status

This calls sets the SPARC, virtualized L2_CONTROL_REG, DRAM, or Zambezi performance register specified by the argument *perfreg*, to the value specified by the argument *value*.

Upon successful completion, it updates the specified performance register *value* and returns a *status* of EOK.

27.5.9.1. Errors

EINVAL	Invalid performance register number
ENOTSUPPORTED	Register number not supported
ENOACCESS	No access allowed to performance registers
EWOULDBLOCK	Cannot complete operation without blocking

27.5.10. UltraSPARC T2+ PCIe performance instrumentation

UltraSPARC-T2+ PCIe performance instrumentation remains unchanged from UltraSPARC T2, see Section 27.4.6, “API calls for PCI-Express interface unit performance counters”.

27.6. UltraSPARC KT performance counters

The UltraSPARC-KT processor is a fully integrated System On a Chip (SOC) design that incorporates processing cores together with memory controllers, a PCI Express IO root complex, and coherency links for multi-node support. Performance instrumentation is provided on-chip for each of SPARC, DRAM, PCI-Express, and Coherency Link sub-systems.

27.6.1. Strand performance instrumentation

Each hardware strand has a pair of registers to control/capture CPU specific instrumentation:

Table 27.9. SPARC performance counters

Description	Access
SPARC Performance Control Register	ASR 0x10
SPARC Performance Instrumentation Counter	ASR 0x11

These registers are directly accessible by the privileged code. The HT bit in SPARC PCR controls the counting of hyperprivileged events, can be set only in hyperprivileged mode. A new sample mode (bit 32 of PCR) is added to the Performance Control Register.

For further information on the register specifications the reader is directed to the UltraSPARC-KT programmers reference manual.

27.6.2. DRAM Performance Instrumentation

Each memory controller has four performance counters, packed into two registers, plus a register to control what is counted. The counters count all events for that memory controller, which receives read and write traffic from eight L2 cache banks.

27.6.3. L2 Cache Control Register

The L2 Control Register adds address interleave ceiling mask and nodeid fields. The PERF_CONFIG bits are the only bits that are exposed through these interfaces. The perf control bits in the L2 Control Register are 37:36 which are bits 1:0 in the virtualized L2 Control Register.

27.6.4. API calls for SPARC and DRAM performance counters

These sun4v APIs provide an interface to read and write the DRAM performance registers as they are not accessible by the privileged software.

The privileged software can use this interface to write the HT bit in the SPARC performance control register as well. The access to write to the HT bit can be denied, in which case the sun4v API returns ENOACCESS. The code using these interfaces must handle such a failure gracefully.

The SPARC performance registers are used to get CLC performance counters by setting the PERF_CONFIG bits in L2_CONTROL_REG. In default mode all L2 misses are counted. The PERF_CONFIG bits in L2 control register can be programmed to count misses serviced from local memory, misses serviced from remote memory or misses serviced by cache-to-cache transfers. As most of the

L2_CONTROL_REG is not accessible by the privilege code, a virtualized generic register is used to program PERF_CONFIG bits in all L2_CONTROL_REG.

The SPARC PCR, the virtualized L2_CONTROL_REG, and the DRAM performance registers are assigned unique performance register numbers (PerfReg#) which uniquely identifies each performance register.

The table below describes the SPARC PCR, the virtualized L2_CONTROL_REG, and the DRAM performance registers.

Table 27.10. UltraSPARC-T3 SPARC, L2, and DRAM performance counters

PerfReg	Node	Description	
0	local	SPARC Performance Control Register	
1	local	All L2 Bank Control Registers	
2+(i*6)	i	MCU 0 Performance Control Register	
3+(i*6)	i	MCU 0 Performance Counter Register 0/1	
4+(i*6)	i	DRAM Performance Counter Register 2/3	
5+(i*6)	i	MCU 1 Performance Control Register	
6+(i*6)	i	MCU 1 Performance Counter Register 0	
7+(i*6)	i	MCU 1 Performance Counter Register 1	

The interface for accessing the SPARC Performance Control Register operates on the entire register and always on the current hardware strand.

27.6.5. kt_get_perfreg

trap# FAST_TRAP

function#	KT_GET_PERFREG
arg0	perfreg
ret0	status
ret1	value

This call reads the value of the SPARC, virtualized L2_CONTROL_REG, or, or Zambezi performance register as specified by the argument *perfreg*.

Upon successful completion the call returns a *status* of EOK and a performance register *value*.

27.6.5.1. Errors

EINVAL	Invalid performance register number
ENOTSUPPORTED	Register number not supported
ENOACCESS	No access allowed to performance registers
EWOULDBLOCK	Cannot complete operation without blocking

27.6.6. kt_set_perfreg

trap#	FAST_TRAP
function#	KT_GET_PERFREG
arg0	perfreg
arg1	value
ret0	status

This calls sets the SPARC, virtualized L2_CONTROL_REG, or DRAM performance register specified by the argument *perfreg*, to the value specified by the argument *value*.

Upon successful completion, it updates the specified performance register *value* and returns a *status* of EOK.

27.6.6.1. Errors

EINVAL	Invalid performance register number
ENOTSUPPORTED	Register number not supported
ENOACCESS	No access allowed to performance registers
EWOULDBLOCK	Cannot complete operation without blocking

27.6.7. API calls for UltraSPARC-T3 PCI-Express performance counters

The following hypervisor API calls provide access to the PCI Express performance counters for a UltraSPARC-T3 processor.

The definition and functionality of the following performance registers is given in the UltraSPARC-T3 Programmer's Reference Manual.

Table 27.11. UltraSPARC-T3 PCI-Express performance counters

PerfReg	Description
0	PEX Performance Counter Select

PerfReg	Description
1	PEX Performance Counter 0
2	PEX Performance Counter 1
3	ATU Performance Counter Select
4	ATU Performance Counter 0
5	ATU Performance Counter 1
6	IMU Performance Counter Select
7	IMU Performance Counter 0
8	IMU Performance Counter 1
9	NPU Performance Counter Select
10	NPU Performance Counter 0
11	NPU Performance Counter 1
12	PEU0 Performance Counter Select
13	PEU0 Performance Counter 0
14	PEU0 Performance Counter 1
15	PEU1 Performance Counter Select
16	PEU1 Performance Counter 0
17	PEU1 Performance Counter 1

27.6.8. kt_ios_get_perf_reg

trap#	`FAST_TRAP`
function#	`KT_IOS_GET_PERFREG`
arg0	devhandle
arg1	perfreg
ret0	status
ret1	value

This call reads the value of the UltraSPARC-T3 IOS performance register specified by the argument *`perfreg`* of the PCI leaf specified by the argument *`devhandle`*.

Upon successful completion, it returns EOK *`status`* and performance register *`value`*.

27.6.8.1. Errors

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

27.6.9. kt_ios_set_perfreg

trap#	`FAST_TRAP`
function#	`KT_IOS_SET_PERFREG`
arg0	devhandle
arg1	perfreg

arg2	value
ret0	status

This call sets the value of the UltraSPARC-T3 IOS performance register as specified by the argument *perfreg* of the PCI leaf specified by the *devhandle* argument to the value specified by the argument *value*.

Upon successful completion, it updates the specified performance register value and returns EOK *status*.

27.6.9.1. Errors

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

Chapter 28. Logical Domain Channel (LDC) infrastructure

28.1. Overview

Logical domain channels (LDCs) are designed as point-to-point communication channels between logical domains or between a logical domain and an external entity such as a service processor or the Hypervisor itself.

Within a LDom a LDC is instantiated as a single endpoint (unless the LDC has been created to loop back to the same LDom). The identity of the owner of the other endpoint is opaque to the LDom - this enables LDCs to be re-connected to other endpoints at will.

Conventional attestation protocols may be layered on top of the basic LDC mechanism if the identity of the owner of the other end of a LDC is required. Such attestation is beyond the scope of this document.

Logical Domain Channels provide two ways of transferring data between endpoints; A simple micro-datagram based transfer mechanism where data is sent in 64-byte packets. The second approach allows clients to export regions of their memory address space to share with clients at the other end of specified LDC connections. The importing clients can then access the remote memory region by either mapping it into its address space, use an Hypervisor API call to copy data to/from exported memory, or program an IOMMU to directly read/write the memory.

28.1.1. Packet based communication

28.1.1.1. Between Domains

Domain-to-Domain LDCs provide clients in each domain a simple message communication mechanism. A domain's LDC transport will register Tx and Rx message queues with the Hypervisor prior for each LDC endpoint on behalf of its virtual device client.

The message queues are very similar to the sun4v cpu_mondo and dev_mondo queues where each entry in the queue holds 64 bytes of data. The transport also uses Hypervisor interfaces to register interrupts for each channel and for targeting these interrupts at specific virtual CPUs.

28.1.1.2. Between Domain and Hypervisor

Domain-to-Hypervisor LDCs provide a way for LDC clients in a domain to communicate with clients in the Hypervisor. Instead of using privileged Hypervisor APIs, LDCs provide a general purpose messaging mechanism that allows clients to send both commands and data as part of messages, and also directly read/write Hypervisor memory. On the domain side, the interfaces are similar to the ones in the case to inter-domain LDCs. The domain client will register a message queue, to transmit and receive packets from the Hypervisor.

Hypervisor clients at the other end of the channel will use an private internal Hypervisor API to register a callback for each endpoint. When a domain sends data, the Hypervisor will invoke the callback registered at the Hypervisor endpoint, to process the LDC packet, in the context of the sending CPU. The Hypervisor will not allocate any internal queues to receive packets from the sending domain. If the internal client, chooses to buffer the incoming datagrams, it may choose to do so by providing its own buffering mechanism.

28.1.1.3. Between SP and Domain/Hypervisor

Communication with the SP over LDCs provide clients in both the guest and hypervisor to send/receive data using LDC APIs. Like domain to hypervisor LDC connections, the interfaces are similar to the ones in the case to inter-domain LDCs. The domain client will register a message queue, to transmit and receive packets from the SP. Hypervisor clients at the other end of the channel will use an private internal Hypervisor API to register a callback for each endpoint. When a domain advances the Tx tail data, the Hypervisor will initiate a send by copying packets out of the Tx queue into the queue associated with that channel in the SRAM.

28.1.2. Shared memory communication

Memory can be shared between domains or between the Hypervisor and a domain using the LDC shared memory framework. The Hypervisor LDC framework provides interfaces to domains that allow them to register tables that contain the list of pages being exported along with its usage criteria and access permissions. The Hypervisor, will then arbitrate access to the exported pages from the importing domains using the tables registered by the exporting domain. The rest of this document will refer to these tables as memory map tables or just map tables.

28.1.2.1. Between domains

At the time of domain initialization, each domain nexus will register with the Hypervisor one or more map tables for each LDC connection. It will also specify the page size for which the table will be utilized. Since each processor MMU has capability to support multiple page sizes, an OS instance and its applications might use different size pages for its memory regions. In the current design, each table will contain entries for pages of one size only. Also since each table is bound to a unique LDC connection, only the domain and client at the other endpoint has implicit access to the pages being exported via this table.

When a client (driver) wants to export memory it will use the nexus API calls to specify the VA range it wants to export. It will need to specify whether the memory being exported is for remote mapping, remote copying or IOMMU access only. The nexus will add entries to the channel's map table and return back to the client a range of cookies that correspond to the VA range. The client driver can then share the cookies with its peer at the other end of the LDC connection.

The driver in the importing domain will then use the cookies it obtained from the exporter to either copy the data to/from of the exported memory, or request the nexus to map the memory associated with the cookies into its address space. In the case of the latter, the nexus will return back to the client driver a RA range(s) that corresponds to the exported memory.

28.1.2.2. Between domain and the Hypervisor

Domain to Hypervisor LDCs can be used to directly read, write, or map Hypervisor memory. Similar to a guest, the Hypervisor can choose to export access to pages in its physical address space to a guest over a LDC connection. It does this by creating a map table that holds the pages it is exporting. It can then provide the guest with a cookie that uniquely identifies the entry in the table. The guest client driver will then use the same interface it uses for domain-to-domain LDCs, to either map or read/write the page in the Hypervisor address space.

28.1.2.3. Between Domain/Hypervisor and the service processor

The LDC infrastructure does not allow exporting memory segments to clients of LDC in the service processor.

28.2. Hypervisor infrastructure

28.2.1. Packet delivery

The Hypervisor provides a simple point-to-point messaging mechanism to send and receive packets over a LDC connection. LDC connections as mentioned earlier allows domains to send data to other domains, or the Hypervisor. The Hypervisor guarantees ordered delivery by creating two locks for packet transfer over LDC.

LDC connections are created by the LDom manager by adding the appropriate nodes in the MD. A guest identifies the LDCs associated with virtual devices by looking in its machine description nodes for the device. Each guest/client registers LDC Tx and Rx queues for each endpoint. A guest initiates a transfer by copying data into its transmit queue and invoking a Hypervisor API to setting the tail for the Tx queue.

If a remote receive queue exists, the Hypervisor sends a interrupt to the remote endpoint signaling it that there is data available for read. The receiving endpoint calls into the hypervisor to read the head and tail for the Rx queue. The hypervisor copies data from the sender's Tx queue to the receiver's Rx queue, and then returns the updated head and tail to the receiver.

28.2.2. Shared memory

This section describes the mechanism by which memory from one logical domain may be exported for access by another logical domain. This facility enables shared memory to be utilized for such functionality as virtual device services.

Using the interfaces described herein, one logical domain may export a number of its own memory pages across a logical domain channel for access and use by the logical domain at the other end of the channel. The mechanism is intended to be directly analogous to the way a domain would export pages of its memory for access by I/O devices on the other side of an I/O bridge (I/O MMU).

28.2.2.1. Map table

The principle means by which a domain may export its local memory across a domain channel is through the use of an export map table that the guest defines within it's own local memory - much like a TSB is used to define local virtual memory mappings. The recipient domain at the other end of the logical channel may make use of the exported memory either by using a hypervisor API call to copy data into or out of its local memory, or by using a hypervisor API call to explicitly map the remote exported memory into its real address space for access.

The real address space of each domain's virtual machine is independent of all the others. Therefore to coordinate references to exported memory between domains, cookies are used to refer to entries within the exporter's map table.

Consider a domain (“X”) that wishes to export a page of memory to another domain (“Y”). For this to be possible a domain channel must connect X to Y. Let us assume that such a channel has been created by the domain manager.

In order to export any memory across this domain channel, domain X must allocate an export map table from its local memory, and assign that map table to its local channel endpoint.

The assigned map table may be used to export multiple pages, which remain exported until explicitly removed from the map table, or the table itself is un-assigned from the channel endpoint.

The map table must be a power of number of entries in size, and must be aligned in memory on a real address boundary equal to its size in bytes. Hypervisor API calls are provided to assign a map table to a

channel endpoint, unassign the table, and to get the table info. A map table may not be assigned to more than one channel endpoint at a time.

28.2.2.2. Map table cookies

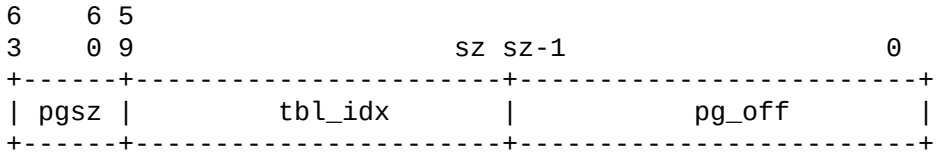
For the recipient domain“Y” to be able to refer to exported memory, it must use a 'cookie' that describes the memory that domain “X” is exporting. This cookie may be considered a form of address for the remote memory, much like a DMA-cookie is used for DMA operations by an IO device.

The export cookie is created by the exporting domain “Y” and it contains two essential pieces of information— the size of the exported page mapping, and the index in the exporter's map table of that mapping. A cookie may also contain offset information so as to identify data located within the memory page defined by a mapping.

A cookie only has meaning within the context of the domain channel its associated map table is bound to. Thus if a map table is assigned to a channel endpoint in domain “X”, then domain “Y” must also identify its local endpoint when using the cookie. In this way the hypervisor is not responsible for creating or tracking or transferring cookies between domains.

A cookie is created by the exporting domain, and can be communicated by any means to the importing domain— for example by message over the same domain channel. When a cookie is used (for example with a `ldc_copy` operation), the associated local channel endpoint enables the hypervisor to determine the remote channel endpoint and the therefore the remote (exporting) domain and the export table itself. The cookie may then be used to locate the entry in the export map table that defines the memory being exported.

Cookies created by an exporting domain have the following format:



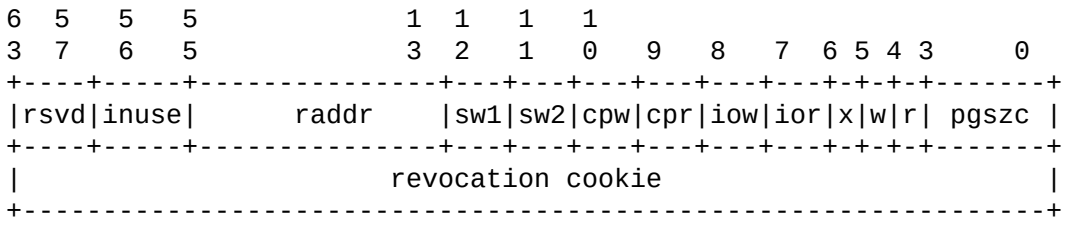
The upper four bits of the cookie identify the page size of the exported page, and use the same page size encodings as the basic sun4v TTE format.

The remainder of a cookie consists of an offset within the specified exported page and an index to the entry within the exporting domain's map table that identifies the actual exported page. The offset field ranges from bit zero, to the number of offset bits relevent for the cookie's page size. The index field starts at the first bit for the page frame number and continues to bit 59. For example, for an 8K page; the page size field (bits 60 to 63) is zero, the page offset is in bits 0 through 12, and the table index is specified in bits 13 through 59.

This compressed cookie format enables a page size, index value and page offset to be transferred in one single 64-bit value that may in effect be treated as an address itself. Basic arithmetic may be applied to the offset field, which if it overflows will automatically adjust the table index field. In this way a large number of sequential map table entries of the same page size can be described by a single cookie value.

28.2.2.3. Map table entry

For the export map table, each entry consists of a two 64-bit words illustrated below:



The map entry (word 0) bit fields are defined as follows:

- Bits 63-57 - reserved - Must be written as zero
- Bit 56 - In use - This bit is set by the hypervisor if a map-table entry is still in use by the importing domain. It is also cleared by the hypervisor if the entry is no longer mapped by the importing domain.
- Bits 55-13 - Real address (RA) - For page sizes larger than 8KB, the low order address bits below the page size must be set to zero
- Bit 12 - SW1 - This bit is available for use by software.
- Bit 11 - SW2 - This bit is available for use by software.
- Bit 10 - Copy writeable (CPW) - If set to 1 the hypervisor `ldc_copy` API may be used by the importing domain to write to this exported page.
- Bit 9 - Copy readable (CPR) - If set to 1 the hypervisor `ldc_copy` API may be used by the importing domain to read from this exported page.
- Bit 8 - I/O Writable (IOW) - If set to 1 this exported page may be mapped by an IOMMU for writing by an I/O DMA operation.
- Bit 7 - I/O Readable (IOR) - If set to 1 this exported page may be mapped by an IOMMU for reading by an I/O DMA operation.
- Bit 6 - e(X)ecute - If set to 1 instructions may be fetched and executed from this page by the importing domain
- Bit 5 - (W)ritable - If set to 1 this page may be mapped and written to as shared memory by the importing domain
- Bit 4 - (R)eadable - If set to 1 this page may be mapped and read from as shared memory by the importing domain
- Bits 3-0 - Page size code (pgszc) - page size code 0 = 8KB, 1=64KB, 2=512KB, 3=4MB, 4=32MB, 5=256MB, 6=2GB, 7=16GB Sizes 8 through 15 are reserved.

The permissions bits (bits 4 through 10) indicate the access permissions granted by the exporting domain to the importer of the page described by the specific map table entry. If no access permissions are granted, (bits 4 through 10 are all zero), the map table entry is considered invalid.

Note: It is recommended that invalid map table entries have the entire 64-bit word set to zero.

Map table entries must not contain overlapping or identical real address ranges— do so yields undefined results for both exporter and importer— without guarantee that the exporter will be able to revoke access permissions to the exported page.

28.2.2.4. Copying in and out of a peer's exported memory

Once a LDC peer has provided access to memory pages via it's map table, a guest operating system can request the hypervisor to copy data into and out of those pages by simply presenting cookies provided by the peer with the `ldc_copy` hypervisor API call.

Each time the call is made the hypervisor validates the presented cookie together with the access permission provided in the exporter's map table to determine whether the copy should indeed be allowed. This is the simplest mechanism by which data may be transferred in bulk between guest operating systems.

28.2.2.5. Mapping page use and restrictions

For a guest to use memory exported by one of it's LDC peers, it must ask the hypervisor to provide access to the exported page. This is achieved using the `ldc_mapin` hypervisor API call.

The map-in call returns a real address of where the imported shared memory page was mapped within the importing guests virtual machine real address space. Shared memory is un-imported using the `ldc_unmap` API call by passing the same real address that was returned from the `ldc_mapin` API call.

As part of the importer's real address space, the imported shared memory page may be used for virtual memory mappings and IO MMU mappings with the same mechanisms as it's own memory pages. However, imported shared-memory pages are not generally accessible like normal memory pages, and the hypervisor enforces a number of restrictions upon their use:

The guest exporting a shared memory page may only allow certain types of access to that page (for example for reading only). For example, attempts to map a page without read or write permission for load or store instructions will fail (or in the case of TSB use generate a data or instruction access exception trap for an invalid real address).

In addition to the restrictions required by the exporting guest, the hypervisor itself requires that importing pages are not aliased either by virtual memory mappings, or IO MMU mappings. Virtual memory mappings are allowed only for context 0 but are available to all virtual CPUs.

Imported shared memory must be unmapped and re-mapped in before a new virtual or IOMMU address may be assigned— even if the old virtual address has been de-mapped with the appropriate demap API call.

28.2.2.6. Mapping revocation

When a guest wishes to discontinue the export of a page to its LDC peer, it can do so by simply denying further access by disabling the access permissions in the map entry word in the corresponding map table entry. (It is recommended that an entry be disabled/invalidated) by writing the value 0 to the whole map entry word (word 0).

Denying future accesses does not automatically revoke existing page mappings to which the LDC peer may have access.

Well-behaved peers sharing exported memory are recommended to use a communication protocol to determine when exported memory pages are available or no longer in use by a peer. It is anticipated, therefore, that only in extraordinary circumstances will a guest that exports memory need to forcibly deny (“revoke”) access to a previously exported memory page.

To avoid the cost of an export revocation for well behaved peers, the hypervisor provides an indication that an exported page is actually still in use by a peer in the form of a revocation cookie in the second word of the map-table entry for the exported page. This revocation cookie word must be initialized to zero when a page is exported, and will be over-written by the hypervisor with a revocation cookie while the exported page is actually in use by the peer guest.

When a page is no longer to be exported, the export mapping permissions should be removed after which the revocation cookie word can be examined to see if the page is actually still in use by the peer guest. A revocation cookie value of zero indicates the page is not in use— at which point the map table entry may be re-used for exporting other pages.

A non-zero value for the revocation cookie indicates that the previously exported page is still in use by the peer guest. It then becomes a matter of policy for the exporter as to whether it wishes to forcibly revoke the access permissions for the importer, or simply wait for the importer to clean-up itself.

To forcibly revoke access permission for the peer guest, the exporting guest simply uses the `ldc_revoke` API call with the LDC cookie for the exported page, and the revocation cookie provided in the export map table.

Removing individual permissions for exported pages must be done by unmapping or revoking access to the exported page first, then re-exporting it with the new permissions required.

Forcibly revoking access to an exported page, can have catastrophic consequences for the importer— including failed memory accesses or failed device DMA transactions. Therefore, the exporter should avoid revocation as far as possible.

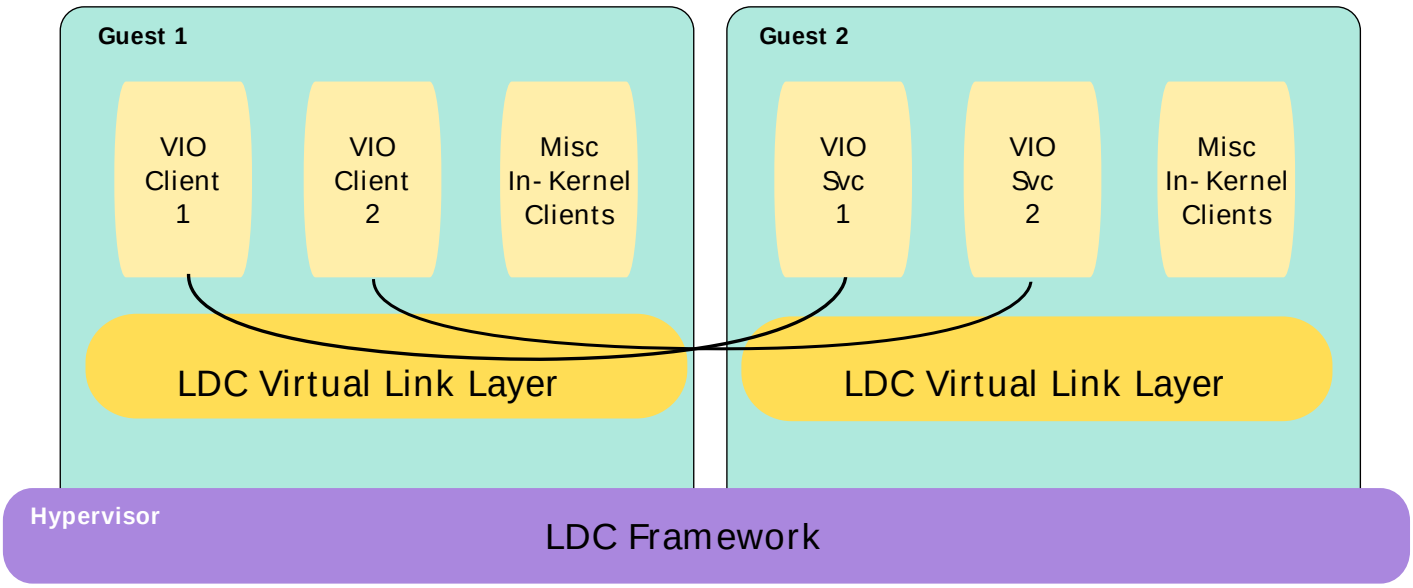
Exit of the exporting guest will cause the hypervisor to automatically forcibly revoke exported page mappings.

An importer of shared memory pages that is intended to be robust should be designed to shield itself against exported mappings being forcibly revoked at any time either by the exporter or automatically by the hypervisor if the exporter exits. Importers wishing to avoid these issues may always use the `ldc_copy` capability to move data.

28.3. LDC virtual link layer

Logical domain channels provide a virtual link layer abstraction that are designed as point-to-point communication channels between logical domains or between a logical domain and an external entity such as a service processor or the Hypervisor itself. Logical domain channels provide an encapsulation protocol onto which higher level transport can be built such as TCP/IP and PPP.

Figure 28.1. LDC Virtual Link Layer



28.3.1. Communication overview

28.3.1.1. Data Transfer Mechanisms

Data transferred between domains can be encapsulated into LDC packets or transferred directly from one domain's memory to another using the Hypervisor shared memory communication support. The link layer protocol defined here provides clients the ability to choose either mechanism for data transfer. The link layer will fragment and reassemble messages as part of the transfer. It will insert additional header information as part of each packet to indicate the start and end of a fragmented data transfer. The LDC link layer uses network byte ordering to transfer all data. The actual details of the transfer protocol itself will be invisible to the clients.

Packet-based Transfer
: Data can be transferred out of a virtual machine by encapsulating it into LDC packets or transferring it directly from one domain's memory to another using the Hypervisor shared memory communication support. The link layer protocol will provide client drivers the ability to choose either mechanism for data transfer.

  In the case of the packet based mechanism, the link layer protocol will fragment and reassemble messages as part of the transfer. It will insert additional header information as part of each packet to indicate the start and end of a fragmented data transfer. The actual details of the transfer itself will be invisible to the client driver. It is recommended that this approach be used only for short messages.

Shared Memory Access
: The shared memory access mechanism allows a client driver to make sections of its memory visible to other domains. This support is build on top of the underlying Hypervisor infrastructure for setting up memory map tables to share memory segments.

  Client drivers will use the interface to obtain a cookie associated with the memory they want to expose. The client can then send the cookie to a client driver in a remote domain using the packet based transfer. The receiving client can then request its LDC framework to consume the cookie and map the remote domain's memory into its address space. Once the mapping is completed, clients can read, write these shared memory regions and also setup DMA operations to directly transfer data into or out of domain buffers.

  A slight modification to the direct memory map is the copy option, where the data is copied in to or out of the buffers that have been exposed by a virtual device client or server via a Hypervisor API. In this approach, when a virtual device wants to send data, either the device client or server will first copy the data from the exporter's memory to a local memory buffer.

Both methods of data transfer is provided because all virtual machine client may not allow shared memory communication either due to technology limitations or security concerns.

28.3.1.2. Protocol Modes

Clients of the LDC mechanism can either be clients that implement sophisticated transport layer like capabilities, i.e. virtual Ethernet with a TCP/IP stack, or a simple client with no special transport capability like the FMA daemon or a virtual console device. These clients have different reliability requirements on the underlying virtual link layer protocol. The virtual link layer protocol will meet the requirements of either type of client by implementing three different types of data transfer protocol.

Raw mode
: The raw virtual link layer protocol protocol does not add any overhead by appending any headers and sends only 64-byte packets at a time. It has no support for session management, message fragmentation

and re-assembly, or retransmissions. It provides a very thin layer over the Hypervisor interface and mostly passes through read and write requests to the Hypervisor.

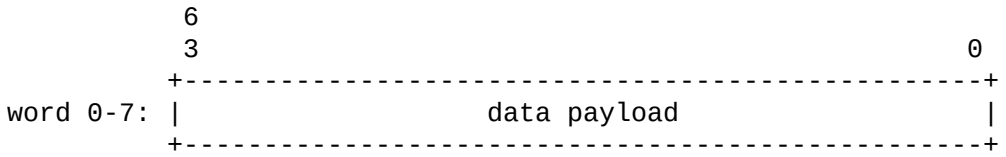
Unreliable mode
: The unreliable link layer protocol will implement a communication mechanism that will include support of connection establishment via a simple handshake protocol. It will also implement support for negotiating a session and detecting session termination. It will only implement support to detect either lost or out-of-order packets, and not reassemble out of order packets and only stitch together packets received in order. The unreliable mode also supports fragmentation and reassembly of LDC datagrams. Clients of this link layer mechanism will need to implement their own error detection mechanism and do the required retransmission.

Reliable mode
: The reliable link layer protocol implements all the support encompassed within the unreliable link layer protocol. In addition, it implements support for streaming buffers, detecting out-of-order packets and packet loss and acknowledges received packets. The primary distinction of reliable mode is to provide an error detection capability via packet ACKs and NACKs.

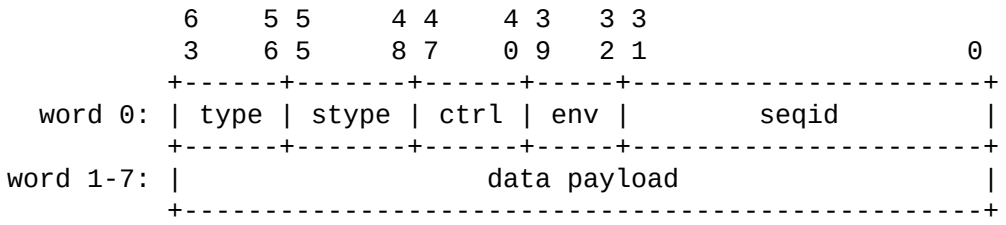
28.3.2. Packet formats

The Hypervisor LDC framework provides the capability to deliver 64-byte packets between peer channel endpoints. It does not impose any predefined format for each word in the 64-byte packet. Depending on whether the clients want to use a raw, reliable or unreliable link mode, the link will utilize different formats for each LDC packet. In the case of the reliable link each packet will consist of a 16-byte header, and 48-bytes of data payload. The unreliable link will have a smaller 8-byte header, and contains 56-bytes of data payload. The raw link will utilize the complete 64-bytes for the data payload. The high-level formats of the raw, unreliable and reliable packet are shown below.

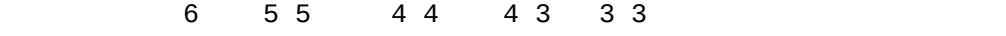
Raw Datagram Packet:

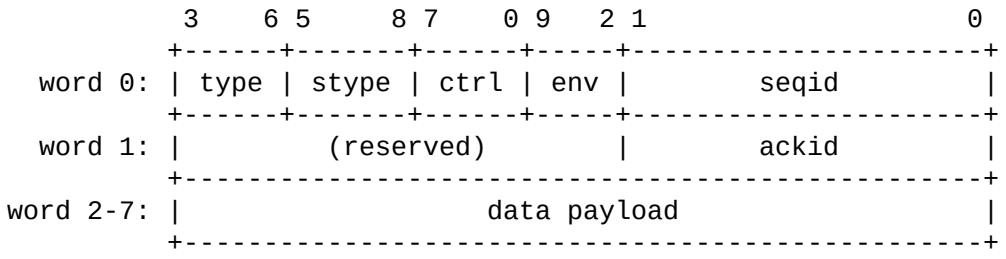


Unreliable Datagram Packet:



Reliable Datagram Packet:





Description:

- Packet Type (Word 0, Bits 0-7): Each packet sent from one LDC endpoint to another can consist of either control, data or error information or a combination there-of. The appropriate 'type' field bit(s) are set to indicate packet contents.

LDC_CTRL	0x01
LDC_DATA	0x02
LDC_ERR	0x10

- Packet Sub-Type (Word 0, Bits 8-15): The stype field contains values INFO, ACK or NACK and defines the type of data, control or error message. The combination of the type and stype fields define the nature of the message.

LDC_INFO	0x01
LDC_ACK	0x02
LDC_NACK	0x04

- Control Info (Word 0, Bits 16-23): The ctrl field contains either basic control information and/or error information. The control info values currently supported are listed below:

LDC_VERS	0x01	Link version
LDC_RTS	0x02	Request to send
LDC_RTR	0x03	Ready to receive
LDC_RDX	0x04	Ready for data exchange

- Packet Envelope (Word 0, Bits 24-31): The env field, depending on the packet type, contains either control or data related information. If the packet contains a control info of type RTS or RTR, the envelope contains protocol mode and will have one of the following values:

LDC_MODE_RAW	0x00	Raw mode
LDC_MODE_UNRELIABLE	0x01	Request to send
	0x02	*Reserved*
LDC_MODE_RELIABLE	0x03	Reliable mode

  When using RAW mode, since there is no handshake as part of the protocol, the RAW mode value specified above is never exchanged as part of the packet envelope. It is only specified here for completeness.

  In the case of packets containing data, the envelope contains the number of bytes in the current packet. It also contains information pertaining to fragmented transfers. The format of the envelope for a data packet is shown below:

```
 3      3     2                             2
 1      0     9                             4
+--------------+-----------------------------+
| stop | start |         pkt_size             |
+--------------+-----------------------------+
```

When a message is fragmented, the first fragment has the *start* bit in the envelope field, set to 1. The last fragment has the *stop* bit set to 1. Intermediate fragments between a start and stop packet have neither bit set. In the case of a single packet transfer (less than the max payload), both *start* and *stop* bits in the envelope are set to 1.

- Sequence ID (Word 0, Bits 32-63): The seqID field is populated with an unique sequential number for every packet sent from one endpoint to another. This is used by the receiver to detect and enforce packet ordering, and acknowledging received packets.

  The AckID field below is only used for the reliable link implementation

  Implementation Note: In order to generate a unique session ID, it is recommended that the link uses 32-bits from the CPU tick register as the session ID.

- Acknowledgment ID (Word 1, Bits 31-63): An endpoint can acknowledge packets it has received by sending an ACK back to its peer. The 'ackid' field contains the sequence ID of the last packet received in correct order by an endpoint. The peer may send separates messages to ACK received packets or embed acknowledgments in data packets.

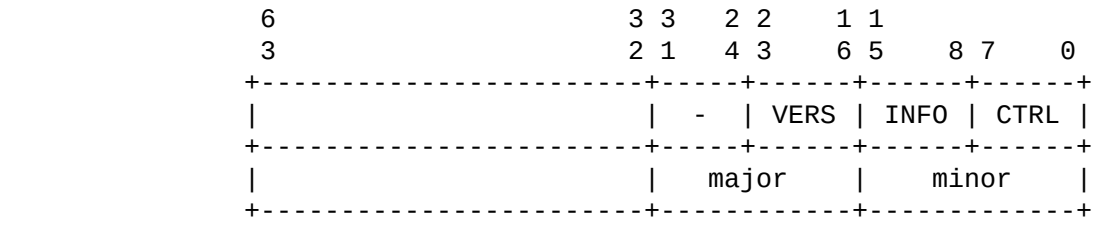
28.3.3. Communication protocol

The link layer implements a thin connection establishment, tear down and data transfer protocol on top of the Hypervisor infrastructure. When clients opens a channel for communication, the link allocates memory for transmit and receive queues and registers these with the Hypervisor. Since neither endpoints have any knowledge about a endpoint's capabilities and whether it is ready to receive data , a simple handshaking protocol is needed to prior to starting the data transfer. This also ensures that clients can start and terminate their sessions independent of each other, and reestablish a connection when necessary.

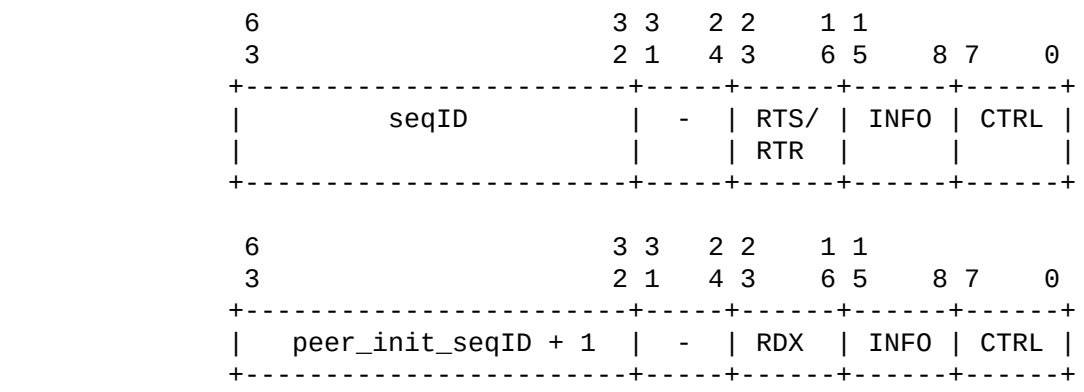
Implementation Note: In the case of a reliable connection, the link should buffer outgoing messages for retransmission purposes. It will mark packets in the transmit queue as completed when it receives ACKs. In the event of a packet loss or timeout, this allows the link to retransmit packets.

28.3.3.1. Session establishment

• After setting up the Tx and Rx queues, either endpoint will initiate a version negotiation by sending a LDC_VERS message, with the version number it supports in the second word of the message. The link will use a simple count down algorithm so that both sides use to agree on a mutual version. If the peer endpoint agrees with the same version or the same major but a lower minor version, it will respond back with an ACK (same msg with the ACK bit set). If it does not support the version, it will respond with an error message NACK and also set the version field to the next lower version version it supports. If it does not support a lower version, it will set the version fields to zero. The sender can then re-send another VERS request with the received lower version or a new even lower version. This will continue on until either the endpoint initiating the VERSION handshake exhausts all the version it supports or the peer accepts a version or responds with an NACK message with version set to zero.



• Following the version negotiation, either endpoints will negotiate a 3-way handshake. As part of this handshake, the endpoints will exchange initial sequence IDs for the session.



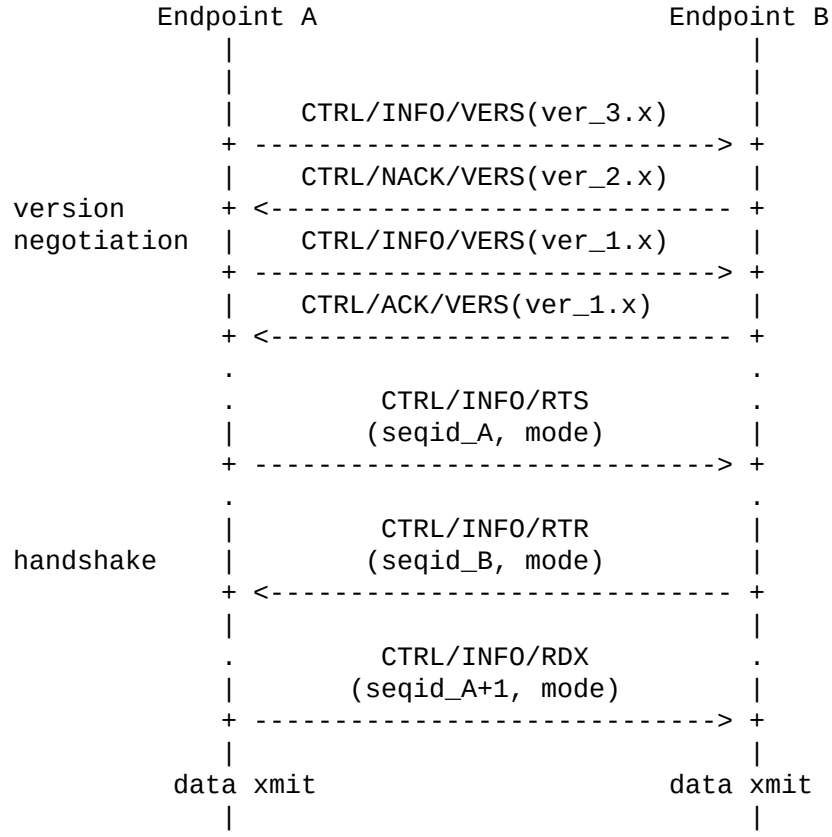
• The sending link endpoint aka endpoint_A will initiate an handshake with the other side i.e. endpoint_B by sending an `LDC_RTS` message that contains the initial seqID (if reliable), and the mode it would like to use for communication.

• If endpoint_B has setup a receive queue, it will either:

• respond back with a `LDC_RTR` message, that contains its initial seqID and the matching link mode message.

• endpoint_A will then respond back with a `LDC_RDX` message. This will mark the channel status as UP and data transfer can now commence.

• If endpoint_B has not setup a receive queue, the hypervisor send (`hv_tx_set_qtail`) operation will fail.



Following a successful handshake, both sides can start transmitting data.

28.3.3.2. Session termination

A session between two endpoints can be torn down either due to a packet error, repeated packet loss, too many retransmissions or at the request of a client. A session is normally terminated by either un-configuring or reconfiguring the receive queue. On receiving a CHANNEL_DOWN or CHANNEL_RESET notification from the Hypervisor the receiver will reset its internal state from which a version negotiation and handshake will need to occur prior to fresh data transmission.

28.3.3.3. Session status notification

A session is established when either endpoints initiate a handshake or is terminated following an Rx queue un-configuration or reconfiguration. Following either events, the link can notify its client about a change in session state via the callback registered by the client.

28.3.3.4. Data transfer

28.3.3.4.1. Packet format

When sending data to its peer, depending on the size, a link will either send the data in one packet or fragment the data into multiple packets. The type field in the msg pkt will be set to DATA for all packet based transfers. The stype field will be of value INFO and the envelope field will contain the number of bytes being sent in each packet. The start and stop bits are used to indicate the start and end of a fragmented transfer. The first packet in the transfer will have the start bit set to 1. Subsequent packets have neither the

start nor stop bit set. The last packet sent as part of a fragmented transfer will have have the stop bit set to 1. If the data is transmitted in a single packet, both the start and stop bit will be set to 1.

28.3.3.4.2. Streaming support

The Reliable mode also implements support for streaming data transfers. It does this by breaking each message into MTU size blocks, specified by the client at the time of channel initialization. During send (`ldc_write`), each message is first broken up into MTU size blocks before being transmitted using the packet transfer approach discussed above. On the receiving end, the link layer passes data back to client in MTU size blocks without any reassembly. Using streaming eliminates the need to allocate very large Tx and Rx queues in the link layer as very large messages can be transferred in MTU size chunks.

28.3.3.4.3. Message ACKs

Message ACKs are used in the case of reliable link mode to indicate data transfer progress.

A client can only queue a fixed number of packets, after which it will have to wait for an ACK from the receiver before it can send more packets. The receiver will periodically respond back with a DATA/ACK control message, and the 'ackid' field will contain the sequence ID of the last packet it received in correct order. Since the packet control field bits for an ACK message do not overlap with those of a regular data packet, a endpoint can send an ACK message embedded in a data packet.

28.3.3.4.4. Transmit queues and retransmissions

In the case of a reliable link, the link will retransmit the packets in the event of a data loss.

For each message sent by a client, the link will maintain it in a list of message segments.

Each segment corresponds to one more fragments i.e. packets in the transmit queue. It will store the seqID corresponding to first fragment with the segment. It will initiate a send by storing the fragmented packets in the transmit queue. At the same time it will start a timer for the message. If a ACK for the packets are not received before the timer expires, the sender will retransmit the message with the same set of start of end seqIDs. If an duplicate ACK is received, it will discard it.

The sender will also maintain a head and tail pointer to keep track of the packets that have been transmitted and the ones that have been ACKed. In the event of a timeout, the sender will retransmit packets by copying over the packets into queue locations starting at tail location. All packets in the queue will purged when a session is torn down and/or established.

There are multiple retransmit scenarios and these are handled in the following manner:

- Packet loss

  This is the simplest of all cases. In the event of packet loss, the receiver will discard all future packets until it receives a packet in correct sequence. The sender will initiate retransmission on timeout.

- Premature timeout / Delayed ACKs

  There are cases when the receiver is backed up and does not respond to the sender in a timely fashion. This will cause the sender to timeout prematurely and retransmit the segment's packets to the receiver. It might either during the retransmission or subsequently receive ACKs for the first transfer. When it receives the ACK, it can mark the message segment as successfully sent. It will then ignore any duplicate ACKs received as a result of the retransmission. Similarly, the receiver will discard packets associated with the retransmission (same seqID range), if it had previously received the message successfully. Even if the receiver discards incoming messages as duplicates, it will need to ACK the messages as earlier ACKs could have been lost.

- Lost ACKs

  In the event, the message was sent successfully, but the ACK was lost, the sender will eventually timeout and retransmit the segment packets. Since receiver already received the message, it will discard the message but still send an ACK. If there is an error during retransmission, the receiver will discard the packets as before.

28.3.3.4.5. Link errors

Either during the initial handshake or during the course of data transmission, either endpoints can detect an error and take the corresponding action. The errors currently detected and handled within the link are listed below:

- Packet error

  During data transmission, packets can either get dropped or gets sent out of order. When the receiver detects a packet that is out of order, it will purge all pending packets in its transmit queue, until it finds a packet with the correct sequence. The unreliable link does not support retransmissions, and packets are dropped on error. Transmit sequence errors are detected via invalid start/stop bits in pkts.

  In the case of reliable link mode, packet loss is detected using seqID. It will send an ACK for the last packet that was received in correct order. This allows the sender to determine what seqID to start the retransmission from. Since there might be packets in flight (pkts between the ACKd pkt and the current TX tail ptr), the receiver will have to continue dropping all future packets until it receives a packet with the seqID that corresponds to the lost packet. The sender will eventually timeout and recopy lost or unacknowledged packets starting from the current tail location and initiate the retransmission of packets starting with the lost packet.

28.3.3.4.6. Link interrupt handler

Links that are capable of handling interrupts can register an interrupt handler for each LDC channel with a target CPU to which the interrupt should be delivered. The link should allocate the CPU to channels in a round-robin manner. When a channel has pending data in its LDC queue, the Hypervisor will send a dev_mondo interrupt to the link. The link will either process the packet in the queue (if it is a control packet), or invoke the client's callback (if it is a data packet) to let it know that there is pending data.

Chapter 29. Virtual IO device protocols

29.1. Virtual IO communication protocol

Virtual devices, clients and/or services, at the most basic level rely on the underlying Hypervisor LDC framework (Chapter 22, *Logical Domain Channel services*) and LDC transport layer (Chapter 28, *Logical Domain Channel (LDC) infrastructure*) to transfer data.

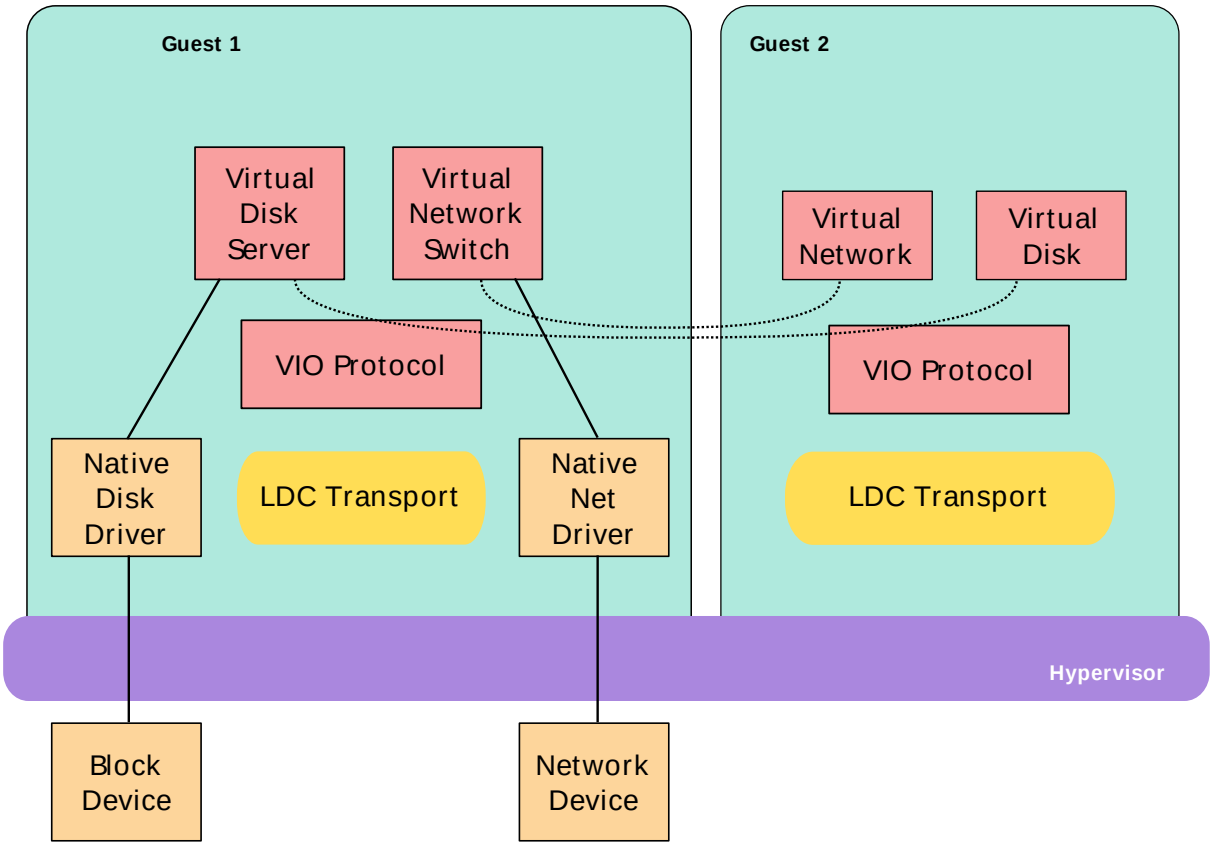
Since both these layers only provide a basic communication mechanism, Virtual IO (VIO) devices employ a basic handshake procedure to agree on transmission properties for the channel, before meaningful data can be exchanged between the two channel endpoints. As part of the handshake they negotiate a common version, device attributes, data transfer type, and if necessary shared memory descriptor ring information. Following a successful handshake, the devices can send and receive data. All VIO devices use the LDC unreliable transport mode for all communication.

The figure below shows two logical domains with VIO device clients and services communicating with each other using the VIO protocol and layered on top of the underlying LDC framework. Domain A has exclusive access to local physical devices through native device drivers and exports access to these devices over the LDC connection to domain B.

29.1.1. VIO data transfer

VIO devices will transfer data either using packet mode by storing the data in LDC datagrams or sharing the data using the shared memory capability of the Hypervisor. A VIO device that uses packet mode, will use either a single LDC datagram packet or use the fragmentation-reassembly capabilities of the LDC transport layer to packetize and transfer larger messages. The Hypervisor shared memory support allows guests to share memory regions in their address space with another guest at the other end of a channel (FWARC/2006/184 [http://arc.opensolaris.org/caselog/FWARC/2006/184/]). This capability allows VIO client drivers to share segments of memory with a VIO client or service so that data can be transferred efficiently and much faster, instead of transferring data over the channel by packetizing each transfer.

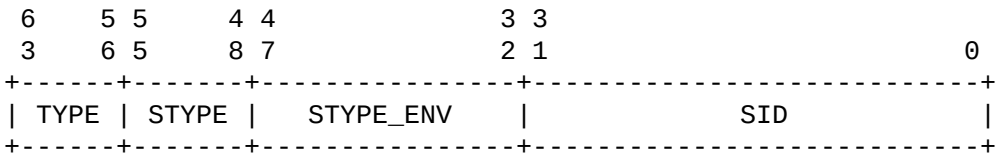
Figure 29.1. Virtual I/O Layers



Like conventional IO devices, the virtual IO devices that use the Hypervisor shared memory infrastructure for data transfer, will setup and use descriptor rings. The descriptor ring is a contiguous circular ring buffer that IO devices use to queue requests, receive responses and transfer associated data. VIO devices that use shared memory will either share their descriptor rings or send the descriptors as in-band messages. The subsequent sections describe the content of control and data packets, the transfer protocol and the structure of the descriptor rings used by VIO devices. It also specifies the device specific content of the LDC packets and descriptors for virtual network and disk devices.

29.1.2. VIO device message tag

All packets exchanged by VIO devices over a channel will use a common message tag as the header for the message. The message tag uniquely identifies the session, the type and subtype of the message. The subtype envelope contains message specific meta-data. All packets sent/received by VIO devices will specify all message tag fields and no field is optional. The format of the message tag along with values for the *type*, *subtype* and *subtype_env* fields are shown below:



Message Types:

VIO_TYPE_CTRL	1
VIO_TYPE_DATA	2
VIO_TYPE_ERR	4

Sub-Message Types:

VIO_SUBTYPE_INFO	1
VIO_SUBTYPE_ACK	2
VIO_SUBTYPE_NACK	4

Sub-Type Envelope:

VIO_TYPE_CTRL	VIO_VER_INFO	0x0001
	VIO_ATTR_INFO	0x0002
	VIO_DRING_REG	0x0003
	VIO_DRING_UNREG	0x0004
	VIO_RDX	0x0005
	reserved	0x0006-0x003f
VIO_TYPE_DATA	VIO_PKT_DATA	0x0040
	VIO_DESC_DATA	0x0041
	VIO_DRING_DATA_REG	0x0042
	reserved	0x0043-0x007f
VIO_TYPE_ERR	*reserved*	0x0080-0x00ff
Device Class-specific sub-type envelopes		
VNET	*reserved*	0x0100-0x01ff
VDSK	*reserved*	0x0200-0x02ff
reserved	*reserved*	0x0300-0xffff

29.1.3. VIO device peer-to-peer handshake

For VIO devices, both the server and/or client has to successfully complete a handshake before data transfer can commence. The handshake can be initiated by either parties. In the description below each message sent or received is specified using the format <type> / <subtype> / <subtype_env>.

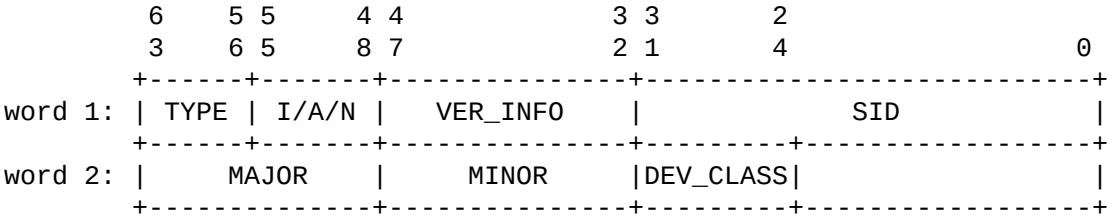
29.1.3.1. Version negotiation

A handshake is initiated by one peer sending a `CTRL/INFO/VER_INFO` to the other endpoint. This message consists of a *`dev_class`* field identifying the type of the sending device, and a *`major`*/*`minor`* pair which specify the protocol version (the protocol version will determine the type and amount of data that will be expected to be exchanged in later phases of the handshake). It also sets the session ID (*`sid`*) to a random value by setting it to the lower 32-bits of the CPU tick. The client will send a new session ID with each version negotiation request. The session ID corresponding to the accepted version gets used as part of each message sent as part of the session.

If the device class is recognized and the version major/minor numbers are acceptable then the receiving endpoint responds back with a `CTRL/ACK/VER_INFO` message leaving all the parameters unchanged. It also stores the sender's SID for use in future message exchanges.

If the major version is not supported, then the peer sends back a `CTRL/NACK/VER_INFO` message containing the next lower major version it supports. If it does not support any lower major numbers, it will NACK with the version major and minor values set to zero. The initiating endpoint can then if it wishes send another `CRTL/INFO/VER_INFO` message either with the major number it received from its peer, if it is acceptable, or with its next lower choice of version. If the major version is supported but not at the specified minor version level, the receiver will ACK back with a lower supported minor version number.

Similarly, if the 'dev_class' is unrecognized, the receiver will respond back with `CTRL/NACK/VER_INFO` with the parameters unchanged and the handshake is deemed to have failed. The format of the version exchange packet is shown below:



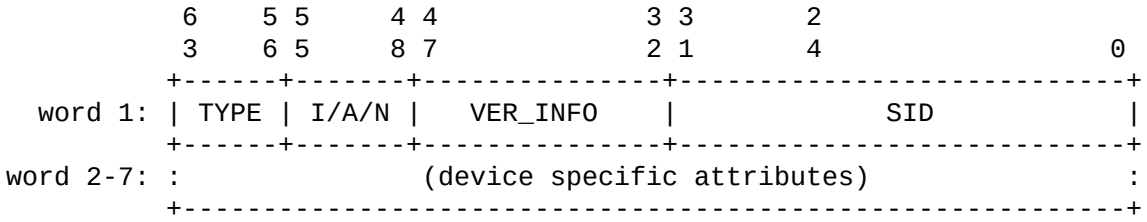
The currently supported devices types are listed below:

VDEV_NETWORK	1
VDEV_NETWORK_SWITCH	2
VDEV_DISK	3
VDEV_DISK_SERVER	4

NOTE: Irrespective of what state the receiving endpoint believes the channel to be in, receipt of a `CTRL/INFO/VER_INFO` message at any time will cause the endpoint to reset any internal state it may be maintaining for that channel and restart the handshake.

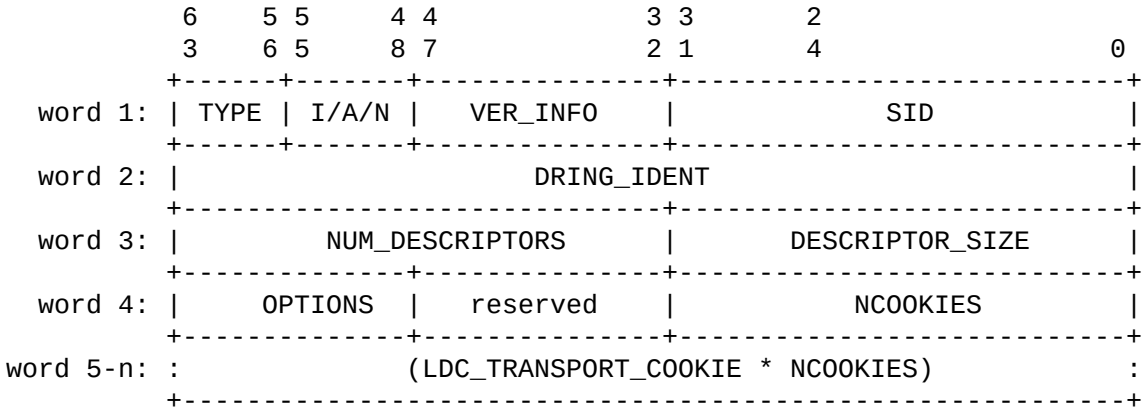
29.1.3.2. Attribute exchange

Following the initial version negotiation phase, VIO device clients/services will exchange device specific attribute information, depending on the device class and the agreed upon API version. Each attribute information packet is of the type `CTRL/INFO/ATTR_INFO` and contains parameters like transfer mode, maximum transfer size, and other device specific attributes. An ACK response is an acknowledgment by the peer that it will use these attributes in future transfer. A NACK response is an indication of mismatched attributes. It is up to the particular device class whether it restarts the handshake or exchanges other attributes. The device specific section for virtual disk and network devices contains more information about the exchanged attributes.



29.1.3.3. Descriptor ring registration

Most virtual devices will use the shared memory capabilities of the Hypervisor LDC framework to send and receive data. Like conventional IO devices, the virtual IO devices will use descriptor rings to keep track of all transactions being performed by the device. Prior to using a descriptor ring, and following version negotiation, and other device specific attribute exchange, VIO clients will register shared descriptor ring information with its channel peer.



A VIO client will register a descriptor ring by sending a `CTRL/INFO/DRING_REG` message to its peer. The message will contain information about the number of descriptors in the ring, the descriptor size, the LDC transport cookie(s) associated with the descriptor ring memory and the number of cookies. The *`options`* field allows certain VIO clients to specify descriptor ring properties that describe its intended use. The *`options`* bits are exclusive and the VIO client must specify only one of the supported values. The supported values in version 1.0 of the VIO protocol are:

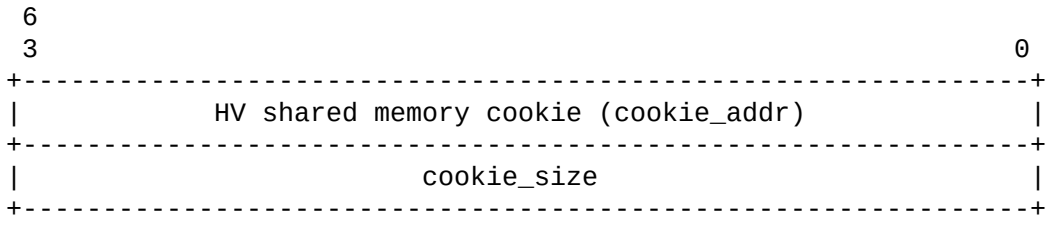
`VIO_TX_DRING`	1	Transmit descriptor ring
`VIO_RX_DRING`	2	Receive descriptor ring

In version 1.6 of the VIO protocol, an additional value is supported:

`VIO_RX_DRING_DATA`	4	Receive descriptor ring with data area

On receiving the registration message, the receiver will ACK the message, and in the ACK provide the sender an unique *`dring_ident`*. The *`dring_ident`* will be used by the sender to either unregister the ring or refer to the descriptor ring during data transfer. A NACK to this message from the receiving end is regarded as a fatal error and the entire session is deemed to have failed and a new session has to be established by re-initiating a handshake. The *`dring_ident`* field is not used in the registration message and only used during the ACK.

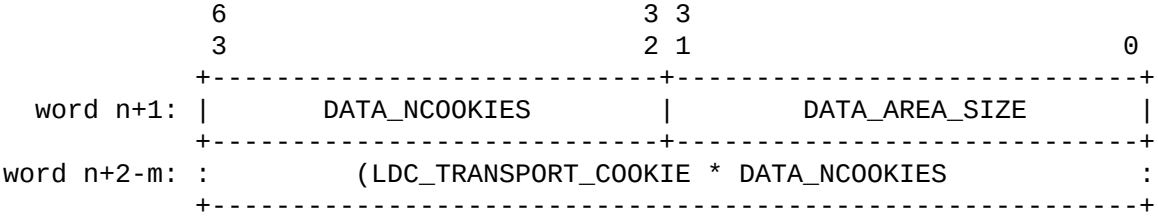
A LDC transport cookie (`LDC_TRANSPORT_COOKIE`) is 16-bytes in size and consists of *`cookie_addr`* and *`cookie_size`* fields. The *`cookie_addr`* field corresponds to the Hypervisor LDC shared memory cookie for each page (see FWARC/2006/184 [http://arc.opensolaris.org/caselog/FWARC/2006/184/]) and the *`cookie_size`* corresponds to the actual number of bytes that is shared within the page pointed to by the cookie. If the descriptor ring memory segment spans multiple pages, an unique transport cookie is used to refer to each page within the segment. The format of the LDC transport cookie is shown below:



When two or more successive pages in the descriptor ring memory segment are stored in consecutive entries in the LDC map table, a single transport cookie can be used refer to all these page entries. The *`cookie_addr`* in this case will still point to first page in the set, but the *`cookie_size`* will correspond to the size spanning all consecutive entries.

A VIO device might typically share multiple descriptor rings with its peer and can choose to register all descriptor rings with its peer at the time of the initial handshake or at any point after data transfer has commenced. If a device intends to do all its data transfer using descriptor rings, it will have to register at least one descriptor ring before data transfer can commence.

With `VIO_RX_DRING_DATA` option, the descriptor ring registration message is extended with additional fields that provide information about the data area that is being exported, as shown below.

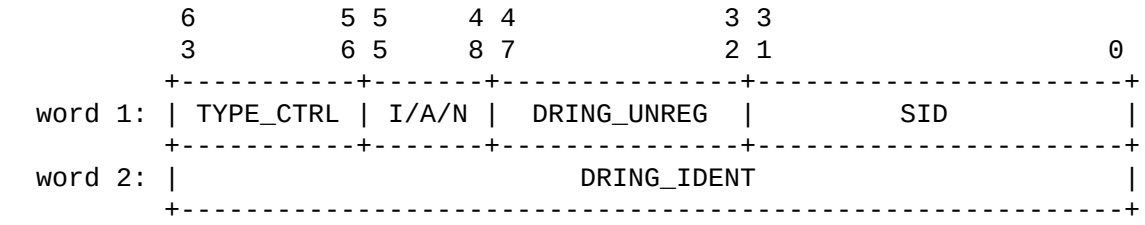


`DATA_NCOOKIES` The number of LDC transport cookies associated with the data buffer area being exported.

`DATA_AREA_SIZE` The size of the data buffer area that is being exported.

In `VIO_RX_DRING_DATA` mode, a VIO device registers a data buffer area with its peer in addition to the descriptor ring. The data buffer area is allocated by the VIO device, as a single large buffer of size *`DATA_AREA_SIZE`*. The data buffer area is managed by the VIO device as individual buffers that are of a certain size determined by the device class specific protocol. For example, in the case of virtual network device class, the buffers may be of size at least equal to the MTU negotiated during attribute phase of handshake. The VIO device exports this buffer area using the hypervisor shared memory infrastructure. It obtains LDC transport cookie(s) to this data buffer area and not the individual buffers. It then passes the cookie(s) to the peer in the `DRING_REG` message. The peer imports this data buffer area using the cookie(s) that it received in the message, using the hypervisor shared memory infrastructure. The peer uses this data buffer area for its data transfers based on the descriptor format specified by its device class.

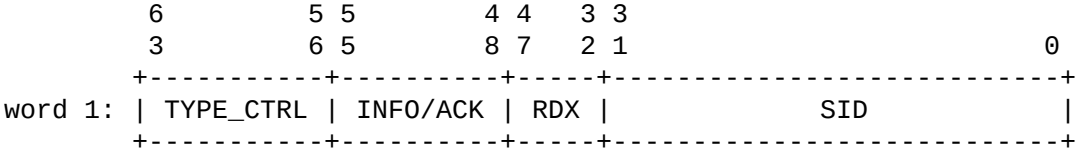
A VIO client can unregister a descriptor ring by sending a `CTRL/INFO/DRING_UNREG` message to its peer. It will specify the *`dring_ident`* it received from the peer at the time of registration. The peer will ACK a successful unregister request and NACK the request if the *`dring_ident`* specified is invalid. If subsequent data transfers refer to an unregistered descriptor ring, the `DRING_DATA` requests will be NACKd.



29.1.3.4. Handshake completion

After successful completion of all negotiations and required information exchange, an endpoint will send a RDX message to its peer to indicate that it can now receive data from it.

An endpoint initiates this by sending a `CTRL/INFO/RDX` message to the receiving end. The receiver acknowledges the message by sending `CTRL/ACK/RDX`. Because LDC connections are duplex, each endpoint has to send a RDX message to its peer before data transfer can commence in both directions. When a RDX is sent by an endpoint, the endpoint is explicitly enabling a simplex communication path, whereby it announces that it can now receive data from its peer. It is VIO device specific whether they require the establishment of a duplex connection before data transfer can commence. There is no payload associated with a RDX message and they are not NACKed.



Once the channel has been established (indicated by the receipt of a RDX message) in either simplex or duplex mode further informational messages may be sent by the initiating endpoint or requested by the receiving endpoint as time goes by. The content and effect these messages have on the session is device specific. These messages are also regarded as in-band notifications.

29.1.4. VIO data transfer modes

VIO devices can send data to their peers over a channel using different transfer modes.

During the handshake, each device will specify to its peer the transfer mode (*xfer_mode*) it intends to use as part of the attribute info message. The device specific attribute message format specifies the location of the *xfer_mode* field in the message. The supported transfer modes in versions 1.0 and 1.1 of the VIO protocol are:

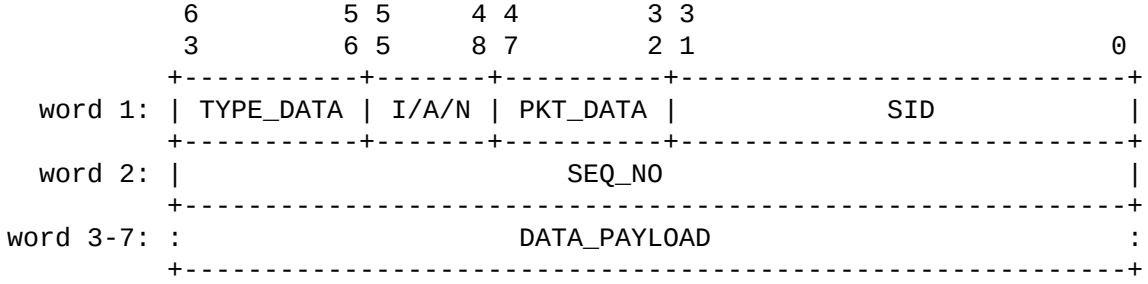
VIO_PKT_MODE	1	Packet-based transfer
VIO_DESC_MODE	2	In-band descriptors
VIO_DRING_MODE	3	Descriptor rings

In version 1.2, the VIO protocol will allow concurrent use of the different transfer modes, specifically packet based transfer and descriptor ring modes. In order to do this, the *xfer_mode* field in the attribute info message will be changed to a bit mask with the following values:

VIO_PKT_MODE	1	Packet-based transfer
VIO_DESC_MODE	2	In-band descriptors
VIO_DRING_MODE	4	Descriptor rings

In version 1.2, the virtual network and switch clients will use the packet transfer mode in addition to the descriptor ring mode (*xfer_mode*=5) to send high priority Ethernet frames as data packets for faster out-of-band processing.

29.1.4.1. Packet based transfer



As discussed in the earlier section, VIO packets always consist of a generic message tag header and a sequence id (which is incremented with each packet sent). Additionally, if a VIO device intends to use packet mode for sending data, it can use up to 40 bytes of a LDC datagram without using LDC transport's packet fragmentation capability. Larger transfers will require the use of the fragmentationreassembly support provided by the underlying LDC transport. The format of a LDC packet containing data is shown above.

29.1.4.2. Descriptor rings

As mentioned in the earlier section, a descriptor ring is a contiguous circular ring buffer VIO devices use to queue requests, receive responses and transfer associated data. Each descriptor in the ring holds request and response parameters specific to the particular device along with opaque cookies that point to the page(s) of memory that are being shared for reading and/or writing. The descriptor ring will utilize Hypervisor shared memory support, so that clients at both ends of the channel can modify the contents of the descriptor(s).

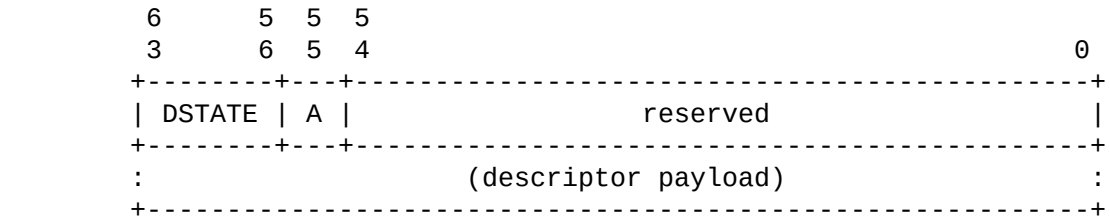
Each VIO client will specify that it intends to use descriptor rings, as part of the attribute info exchange. It will also specify whether or not it intends to share the descriptors using shared memory or send each descriptor as an in-band message. If it shares the descriptor ring using shared memory, it will register at least one descriptor ring with its peer at the other end.

29.1.4.2.1. Descriptor format in VIO_RX_DRING_DATA mode

If the dring mode option chosen between VIO devices is `VIO_RX_DRING_DATA`, the format of the descriptor is device class specific. Currently, it is defined for only the virtual network class; Section 29.3.2, “vNet descriptors” contains more information on this.

29.1.4.2.2. Descriptor format in VIO_TX_DRING/VIO_RX_DRING mode

Each entry in a descriptor ring consists of a common descriptor ring entry header and the descriptor payload as shown in the figure below. The descriptor payload consists of fields that are device class specific and are discussed in more detail in Section 29.2, “Virtual disk protocol” and Section 29.3, “Virtual network protocol”.



The descriptor dstate specifies the state of the the descriptor. The valid state values are:

`VIO_DESC_FREE`	`0x01`
`VIO_DESC_READY`	`0x02`
`VIO_DESC_ACCEPTED`	`0x03`
`VIO_DESC_NONE`	`0x04`

Initially when a descriptor ring is allocated, all entries in the ring are marked with value of `VIO_DESC_FREE`. When a client queues one or more requests, it will change the flags value for the corresponding descriptor(s) to `VIO_DESC_READY`. It will then send a message to its peer requesting it to process the descriptors. The client that is processing the descriptor will first change the state to `VIO_DESC_ACCEPTED`, acknowledging receipt of the request and prior to processing the request.

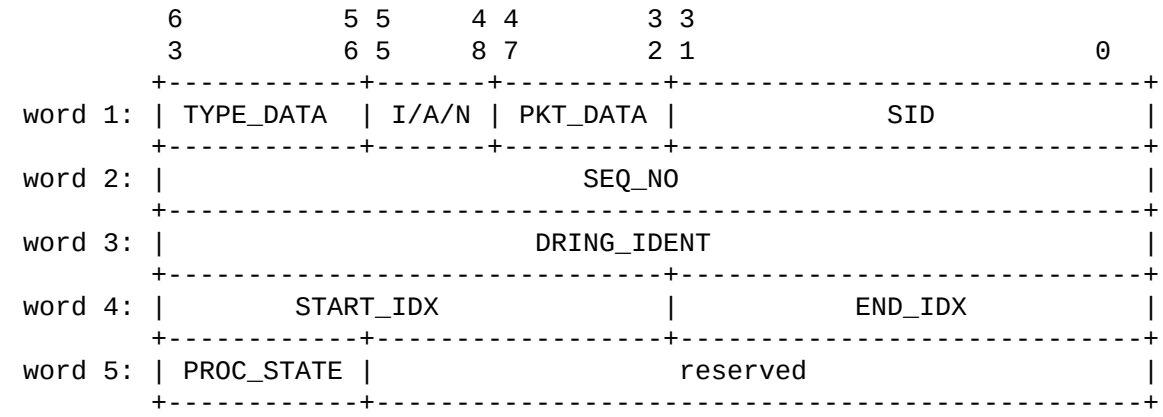
On completing the request, it will update the descriptor with its response and change the value of the flag to `VIO_DESC_DONE`. The client that initiated the request, will take the appropriate action after seeing the request as been marked as `VIO_DESC_DONE` and then change it to `VIO_DESC_FREE`. If the state of a descriptor transitions to an unexpected state, the behavior is undefined. A VIO device under these circumstances, might either reset the session and restart the handshake, or send an error message to its peer.

29.1.4.2.3. Descriptor Ring Data Message Format (common to all dring modes)

When the requesting client updates one or more descriptors and marks them as ready for processing, it will send a `DATA/INFO/DRING_DATA` message to its peer at the other end of the channel. The message will contain the *`dring_ident`* the requester received at the time of registering the descriptor ring. It also specifies the start and end index corresponding to the descriptors that have been updated. If end index value specified is -1, the receiver will process all descriptors starting with the start index and continue until

it does not find a descriptor marked VIO_DESC_READY. The receiver at this point will send an implicit ACK to the sender to let it know that it is done processing all requests. Subsequently, if the sender marks additional entries as VIO_DESC_READY, it will reinitiate processing by sending another DRING_DATA request.

If the start and end index, either overlap with requests sent earlier or correspond to descriptors not in VIO_DESC_READY state, the request will be NACKed by the receiver.



The requester can also request an explicit acknowledgment from the client processing the request (to track progress) by setting the (A)cknowledge field in the descriptor. The client, after processing the descriptor (changes state as VIO_DESC_DONE), will send a DATA/ACK/DRING_DATA message with the *dring_ident* for this descriptor ring and *end_idx* equal to this descriptor.

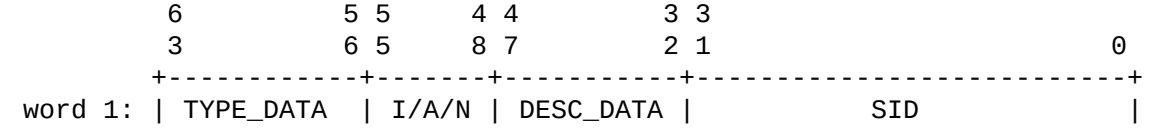
When the requester sends requests with an *end_idx* = -1, the *proc_state* field in the ACK/NACK message, is used by the receiver to indicate its current processing state. The valid *proc_state* field values are:

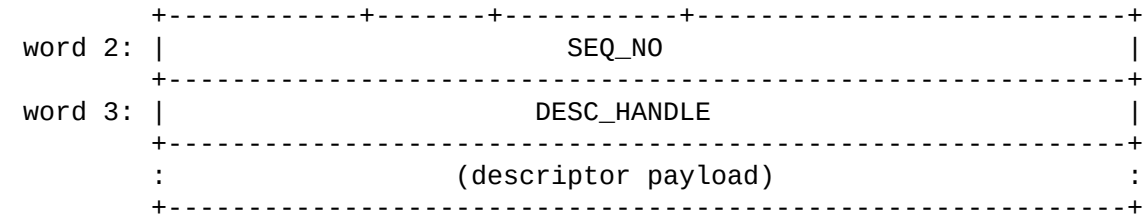
VIO_DP_ACTIVE_MODE	1	Active processing req
VIO_DP_STOPPED	2	Stopped processing req

If the receiver continues to process requests or is waiting for more descriptors to be marked VIO_DESC_READY, it will ACK with *proc_state* set to VIO_DP_ACTIVE. Instead, if the receiver stops after processing the last ACK/NACK, and is waiting for an explicit DATA/INFO/DRING_DATA message, it will set the *proc_state* set to VIO_DP_STOPPED.

The *proc_state* value is then used by the requester to determine when the receiver's state, and accordingly sends an explicit DRING_DATA message when more requests are queued. It is not always necessary that clients need to register a shared descriptor ring to make use of the HV shared memory infrastructure. A simpler client can still use the shared memory capabilities and instead of sharing the descriptor ring, it will send the descriptor itself as in-band data. The DESC_HANDLE in the pkt is an opaque handle that corresponds to the descriptor in the sender's ring.

The content of the in-band descriptor packet is shown below:





In case of both a DRING_DATA and DESC_DATA message, if the receiver gets a data packet out of order (as indicated by a non-consecutive sequence number) then it will NACK the packet and will not process any further data packets from this client. If there are no errors the receiver will ACK the receipt of descriptor ring or descriptor data packets if there is an explicit request by the sender to ACK a data packet by setting the (A)cknowledge bit in the descriptor.

Implementation Note

Upon receipt of a NACK, the sending client can either try to recover or stop sending data and return to initial state and restart the channel negotiation again.

For virtual network and virtual switch devices, in version 1.6 of the VIO protocol, if the dring mode negotiated is RX_DRING_DATA, some of the fields in the DRING_DATA message are interpreted differently:

- The *seq_no* field serves as only an unique ID for the packet. The sender may not guarantee that the DRING_DATA messages (INFO/ACK/NACK) will be sent with the *seq_no* in order.
- The receiver may specify a start index of -1 in its ACK message, to indicate that the sender should ignore the start index of range of descriptors being ack'd and only the end index (last processed descriptor index) is valid.
- The ACK bit in the descriptor is reserved (See Section 29.3.2, “vNet descriptors”) and is ignored if specified by the sender. Thus a peer may not send ACK/NACK messages with a *proc_state* value of VIO_DP_ACTIVE.

29.1.5. Virtual IO Dynamic Device Service (DDS)

Virtual IO devices following the initial handshake, send and receive data using the packet and/or descriptor based modes as described in the earlier sections. This forms the under pinnings of the virtual IO data transfer infrastructure in a LDoms environment. While compelling for a variety of application workloads, virtualized I/O still does not provide high performance I/O capabilities that certain I/O oriented workloads require. The Hybrid I/O model provides the opportunity to share device resources across multiple client domains with better granularity while overcoming the performance bottlenecks of virtualized I/O.

A new control message type will be added in VIO protocol versions 1.3 and higher to support the Hybrid IO model. The new Dynamic Device Service (DDS) control message, with a subtype envelope value of VIO_DDS_INFO, will provide virtual IO devices and services the ability to exchange and share physical device resource information with their peers.

VIO_DDS_INFO	6	DDS information

Each DDS control message will allow a device to share or reclaim a resource, or change the properties of a resource. A peer on receiving a CTRL/INFO/DDS_INFO message, will take necessary action and then either ACK or NACK the message depending on whether the requested operation was successful or not.

Each VIO_DDS_INFO message, in addition to the VIO msg header, includes a DDS message header consisting of a DDS class, subclass, and *request_id* fields. Though the format of the DDS message

header itself is generic to the VIO protocol, the DDS message class and sub-class values are specified by the virtual network or disk devices. The DDS request ID in the header will used to correlate the INFO requests with ACK and NACK responses. The DDS msg format is shown below:

```
          6          5 5        4 4         3 3
          3          6 5        8 7         2 1                          0
         +------------+----------+-----------+----------------------------+
  word 1: | TYPE_DATA  |   I/A/N  | DDS_INFO  |           SID              |
         +------------+----------+-----------+----------------------------+
  word 2: |   CLASS    | SUBCLASS | reserved  |        DDS_REQUEST_ID      |
         +------------+----------+-----------+----------------------------+
word 3-7: :                  (dds message payload)                         :
         +-------------------------------------------------------------------+
```

Device specific class and subclass values, including contents of the DDS message is discussed in Section 29.3.4, “Network Device Resource Sharing via DDS”. The class value ranges reserved for various VIO device classes is specified below:

`DDS_GENERIC`	`0x00-0x0f`	Generic DDS class
`DDS_VNET`	`0x10-0x1f`	Generic DDS class
`DDS_VDSK`	`0x20-0x2f`	Generic DDS class
reserved	`0x30-0xff`	Reserved

29.2. Virtual disk protocol

In the protocol outlined above, the attribute exchange and descriptor payload contents are undefined and left to be specified by the VIO devices. This section describes the contents of these packets for use by both the virtual disk client and server to exchange data. The vDisk client, following an attribute exchange, will send to the server block disk read and write requests, in addition to disk control requests. The server will export each block device over an unique channel, and accept requests from the client, once a session has been established.

29.2.1. Attribute information

During the initial handshake, as part of the `CTRL/INFO/ATTR_INFO` message, the virtual disk server and client exchange information about the transfer protocol and the physical device itself. The format of the attribute contents is shown below:

The vDisk client will provide the server with the transfer mode (*`xfer_mode`*) and the requested maximum transfer size (*`max_xfer_sz`*) it intends to use for sending disk requests to the server.

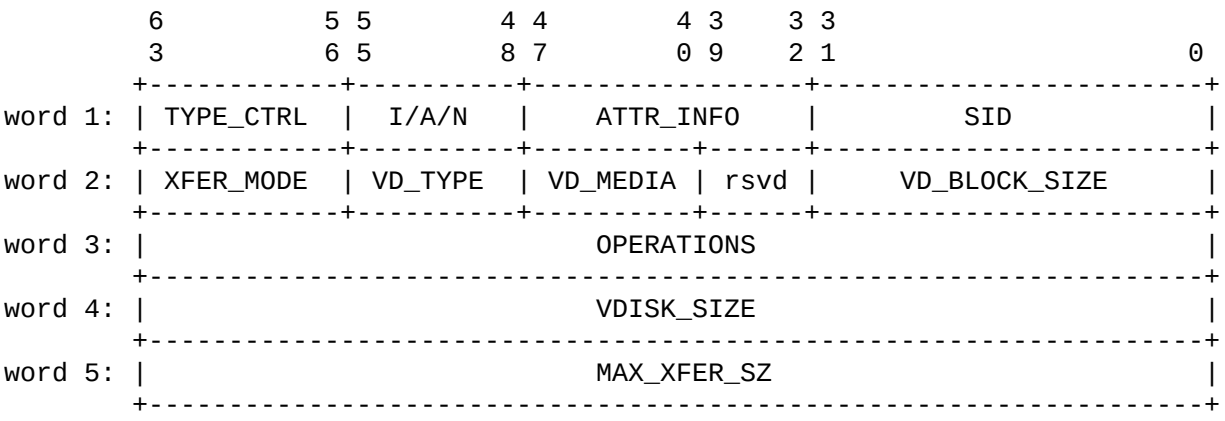
The *`vdisk_block_size`* is specified in bytes. The *`vdisk_size`* and *`max_xfer_sz`* are specified in multiples of the *`vdisk_block`* size.

For version 1.0 of the vDisk protocol the client's request must set *`vdisk_block_size`* to the minimum block size the client wishes to handle, and specify the *`max_xfer_size`*. If the server cannot support the requested *`vdisk_block_size`* or *`max_xfer_sz`* requested by the client, but can support a lower size, it will specify its *`vdisk_block_size`* and/or a lower *`max_xfer_sz`* in its ACK. If the client has no minimum block size requirement it may use the value of 0 as its requested *`vdisk_block_size`*, in this case the *`max_xfer_size`* in the client's attribute request to the server is interpreted as being specified in bytes. Either client or server may simply reset the LDC connection if they fail to agree on communication attributes.

For version 1.1 of the vDisk protocol, the vDisk server can set *vdisk_size* to -1 if it can not obtain the size at the time of the handshake. This can happen when the underlying disk has been reserved by another system. Under these circumstances, the vDisk client can retrieve the size at a later time, after the completion of the handshake, using the VD_OP_GET_CAPACITY operation.

If either client or server cannot support the specified transfer mode, the connection will be reset and the handshake may be restarted. The server in its ACK message will also provide the vdisk type (*vd_type*), *vdisk_block_size* and *vdisk_size* to the client. The supported types are:

VD_DISK_TYPE_SLICE	1	Slice in a block device
VD_DISK_TYPE_DISK	2	Entire block device



All other disk types are reserved and for version 1.0 of the vdisk protocol should be considered as an error.

Only in protocol versions 1.1 and higher of the vdisk protocol, the server in its ACK message will provide the client the *vdisk_size* (specified as a multiple of the block size), and the vdisk media type (*vdisk_mtype*). The supported vdisk media types are:

VD_MEDIA_TYPE_FIXED	1	Fixed device
VD_MEDIA_TYPE_CD	2	CD device
VD_MEDIA_TYPE_DVD	3	DVD device

All other disk media types are reserved and for version 1.1 of the vdisk protocol should be considered as an error.

Both these fields are reserved and not available in version 1.0 of the vdisk protocol. Clients should use the disk geometry information (see Section 29.2.3.9, “VDisk Get Disk Geometry (VD_OP_GET_DISKGEOM)”) to compute the vdisk size.

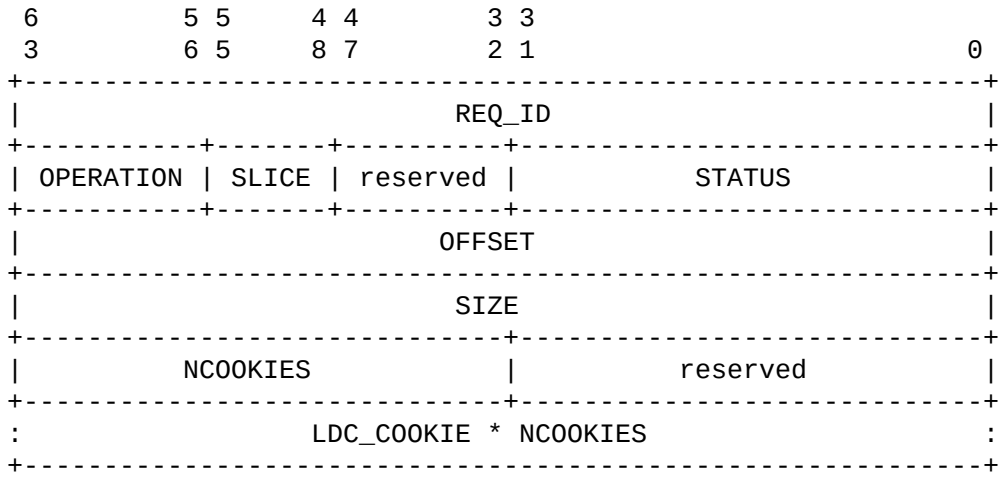
The operations field is a bit-mask specifying all the disk operations supported by the server, where each bit position, if set, corresponds to the operation command supported by the server. The list of supported operations encodings is described in Section 29.2.3, “Disk operations”.

29.2.2. vDisk descriptors

Virtual disk clients will send their disk requests by queueing them in descriptors as part of a shared descriptor ring.

As requests are initiated only by the client, and the buffers pointed to by each descriptor are used for both writing and reading disk blocks, the vDisk client will register the descriptor ring as both a Tx and Rx ring. In the case of descriptor rings that are not shared, the virtual disk client will send the requests as in-band descriptor messages.

The descriptor payload is formatted as follows:



The payload contains the operation being performed.

The *`offset`* field specifies the relative disk block address when doing a block read or write operation to the disk. This corresponds to the block offset from the start of the disk, or the disk slice as appropriate. It is specified in terms of the *`vdisk_block_size`* received from the server. The size field specifies the number of blocks being read or written when doing a `VD_OP_BREAD` or `VD_OP_BWRITE` operation. In the case where the *`vdisk_block_size`* in the client's attribute request is zero the size is interpreted as being specified in bytes.

29.2.3. Disk operations

For each client request sent to the server, the server will process the descriptor contents and submit the request to the device. Each virtual disk request is identified by an unique *`req_id`*. The *`operation`* field specifies the operation being done on the device. The server will then return the status of the operation in the same descriptor but with the *`status`* field containing the outcome of the operation. The supported values in version 1.0 of the vdisk protocol are:

`VD_OP_BREAD`	`0x01`	Block read
`VD_OP_BWRITE`	`0x02`	Block write
`VD_OP_FLUSH`	`0x03`	Flush disk contents
`VD_OP_GET_WCE`	`0x04`	Get write cache status
`VD_OP_SET_WCE`	`0x05`	Enable/disable write cache
`VD_OP_GET_VTOC`	`0x06`	Get VTOC
`VD_OP_SET_VTOC`	`0x07`	Set VTOC
`VD_OP_GET_DISKGEOM`	`0x08`	Get disk geometry
`VD_OP_SET_DISKGEOM`	`0x09`	Set disk geometry

VD_OP_GET_DEVID	0x0b	Get device ID
VD_OP_GET_EFI	0x0c	Get EFI
VD_OP_SET_EFI	0x0d	Set EFI
reserved	0x0a,0x0e-0xff	Reserved for 1.0

In addition, the following values are supported in version 1.1 of the vDisk protocol:

VD_OP_SCSICMD	0x0a	SCSI control command
VD_OP_RESET	0x0e	Reset disk
VD_OP_GET_ACCESS	0x0f	Get disk access
VD_OP_SET_ACCESS	0x10	Set disk access
VD_OP_GET_CAPACITY	0x11	Get disk capacity
reserved	0x12-0xff	Reserved for 1.1

As mentioned before, the vDisk server at the time of the initial attribute exchange will specify the bit mask of operations it supports. If the server does not support a required operation, it is up to the specific client implementation to decide whether it returns an error or internally implements the operation. All operations can be optionally implemented by a particular vDisk server implementation.

If an operation is supported by the server, the outcome of the operation will be always available in the descriptor ring entry *status* field.

The *ncookies* and *ldc_cookie* fields refer to the segment of memory from/to which data is being read/written. See Section 29.1.3.3, “Descriptor ring registration” for more information about the LDC transport cookie.

29.2.3.1. Disks and slices

A vdisk server may export either an entire disk device, or a simple slice (or partition) of a disk to a client as configured by the administrator. In the event that an entire disk is exported to a client, it is client policy as to how it determines the partitioning information or re-partitions that whole virtual disk.

To enable a server to potentially mount or examine a disk created by a client, the server may elect to offer the VD_OP_GET/SET_VTOC operations to its client. If the client elects to use these operations to retrieve partition information, the client when it reads or writes to the disk must specify the slice being accessed — in this case the *offset* field for those transactions is specified relative to the start of the referenced slice (not the start of the disk).

A client is not required to use the VTOC operations, and the server is not required to support them. In either of these events, if the client wishes to use the disk exported by the server it must read (and write, if re-partitioning) its own partition table at some client specific location on the disk.

Attempts to mix reads and writes with get and set VTOC operations to read/manipulate disk partition information have undefined results, and clients are required (though this may only be optionally enforced by the server) to use a consistent approach to discovering or modifying disk partition information.

The slice field is currently only used for VD_OP_BREAD and VD_OP_BWRITE. For all other operations it is ignored, and should be set to zero. If the disk served is of type VD_DISK_TYPE_SLICE the slice field is treated as reserved; i.e. must be set to zero, and ignored by the consumer. For a VD_DISK_TYPE_DISK the slice field refers to the disk slice or partition on which a specific operation is being done — the field only has meaning for disk servers that export a GET_VTOC service so that clients know which slice corresponds to which partition.

If the vDisk client does not use the VTOC service, it must specify a value of 0xff for the slice field for read and write transactions so that the server knows that the offset specified is the absolute offset relative to the start of a disk. Mixing read and write transactions to specific slices together with absolute disk transactions has undefined results, and clients must not do this. A client must close the disk channel and re-negotiate the vDisk service if it wishes to switch between using slice based access (explicitly passing the value of the slice being accessed) and absolute access (where slice is 0xff) when the server offers a disk type of `VD_DISK_TYPE_DISK`.

29.2.3.2. VDisk Block Read command (`VD_OP_BREAD`)

This command performs a basic read of a block from the device service. The decriptor ring entry for this command contains the offset and number of blocks to read together with the LDC cookies for the data buffers.

Once completed the status field in the descriptor is updated with the completion status of the operation.

29.2.3.3. VDisk Block Write command (`VD_OP_BWRITE`)

This command performs a basic write of a block from the device service. The decriptor ring entry for this command contains the offset and number of blocks to write together with the LDC cookies for the data buffers.

Once completed the *`status`* field in the descriptor is updated with the completion status of the operation.

29.2.3.4. VDisk Flush command (`VD_OP_FLUSH`)

This command performs a barrier and synchronisation operation with the disk service.

There are no additional parameters in the decriptor entry for this command.

Before completing this command, the disk service will ensure that all previously executed write operations are flushed to their respective disk devices, and all previously executed reads are completed and their data returned to the client.

29.2.3.5. VDisk Get Write Cache enablement status (`VD_OP_GET_WCE`)

This command is used by a virtual disk client to query whether write-caching has been enabled on the disk being exported by the vDisk server. The payload is a single 32 bit unsigned integer. A value of 0 means write caching is not enabled, a value of 1 means write-caching is enabled (a flush operation should be used as a barrier to ensure writes are forced to non-volatile storage). All other values are reserved and have undefined meaning.

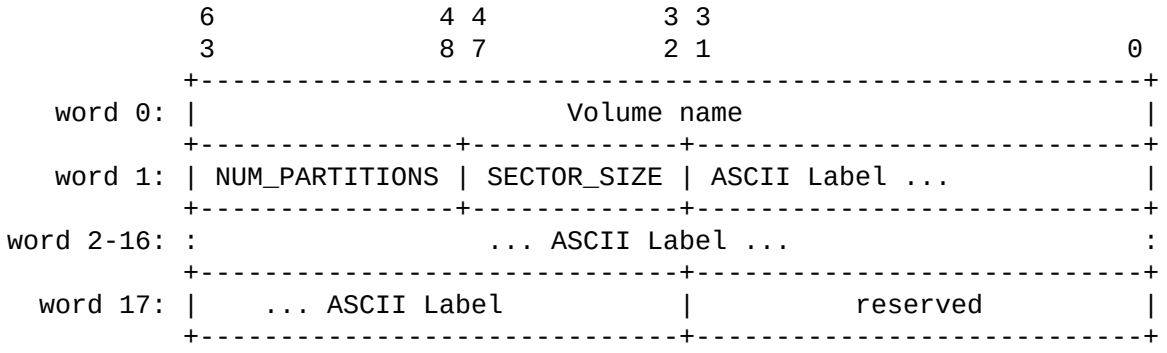
29.2.3.6. VDisk Enable/Disable Write Cache (`VD_OP_SET_WCE`)

This command is used a virtual disk client to enable or disable the write cache on the disk being exported by the vDisk server. The payload is a single 32-bit integer. A value of zero disables writecaching on the server side. A value of one enables write caching on the server side. All other values are reserved and are treated as errors by the vDisk server.

29.2.3.7. VDisk Get Volume Table of Contents (`VD_OP_GET_VTOC`)

This command is used to return information about the table of contents for the disk volume a client is attached to. The successful result of this command includes the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.

The returned data structure has the following header format:

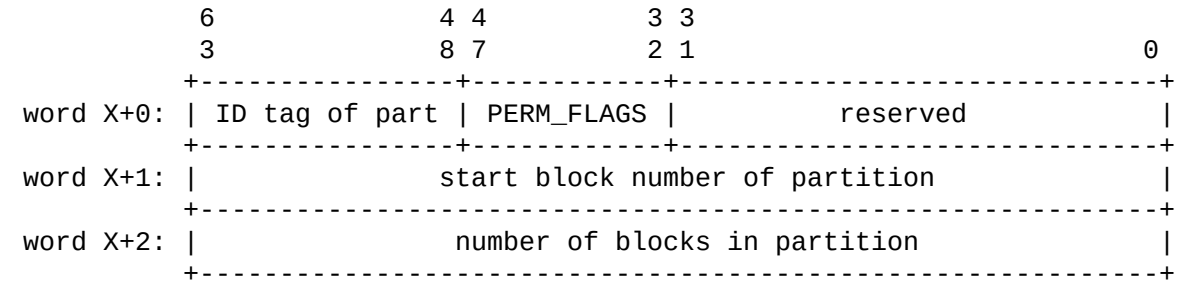


The volume name is an 8 character ASCII name for the volume.

The ASCII label is a 128 character ASCII label assigned to this disk volume. This is distinct from the actual volume name.

The field *sector_size* is the size in bytes of each sector of the disk volume.

The field *num_partitions* is the number of partitions on this disk volume. The header described above is immediately followed by the structure below repeated once for each of the number of partitions specified by the header:



Reserved fields should be ignored.

29.2.3.8. VDisk Set Volume Table of Contents (VD_OP_SET_VTOC)

This command is used by a virtual disk client to set the table of contents for the disk volume the client is attached to.

The supplied data structure has the same format as for the get VTOC command (VD_OP_GET_VTOC). Reserved fields must be set to zero.

29.2.3.9. VDisk Get Disk Geometry (VD_OP_GET_DISKGEOM)

This command is used to return the geometry information about the disk volume a client is attached to. The successful result of this command includes the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.

The returned data structure has the following format:

Offset	Size	Name	Description
0	2	*ncyl*	Number of data cylinders
2	2	*acyl*	Number of alternate cylinders
4	2	*bcyl*	Cylinder offset for fixed head area
6	2	*nhead*	Number of heads
8	2	*nsect*	Number of sectors
10	2	*intrlv*	Interleave factor
12	2	*apc*	Alternate sectors per cylinder (SCSI only)
14	2	*rpm*	Revolutions per minute
16	2	*pcyl*	Number of physical cylinders
18	2	*write_reinstruct*	Number of sectors to skip for writes
20	2	*read_reinstruct*	Number of sectors to skip for reads

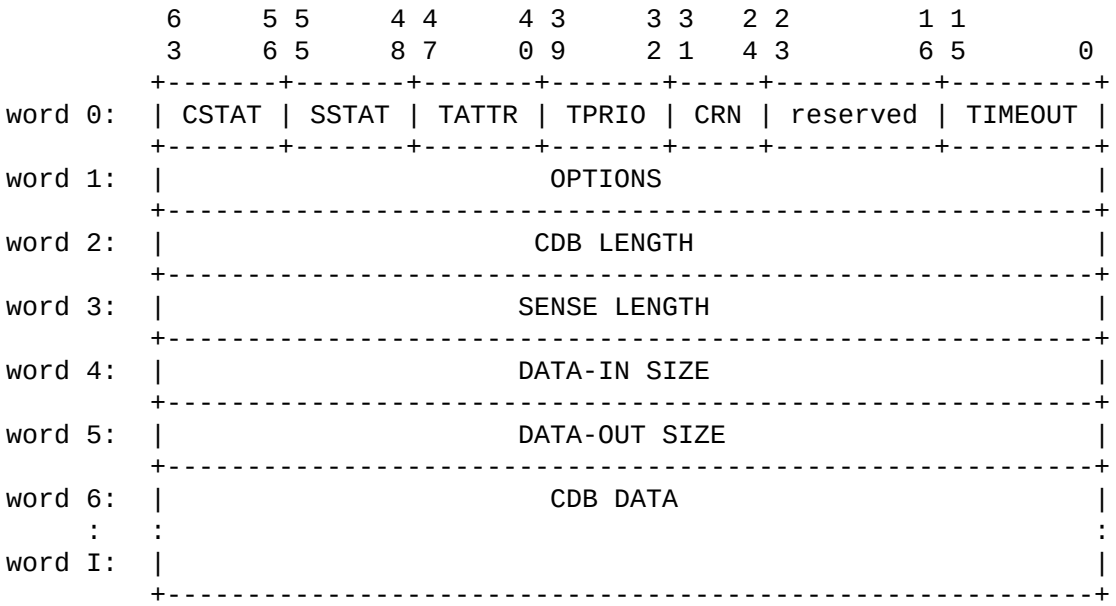
29.2.3.10. VDisk Set Disk Geometry (VD_OP_SET_DISKGEOM)

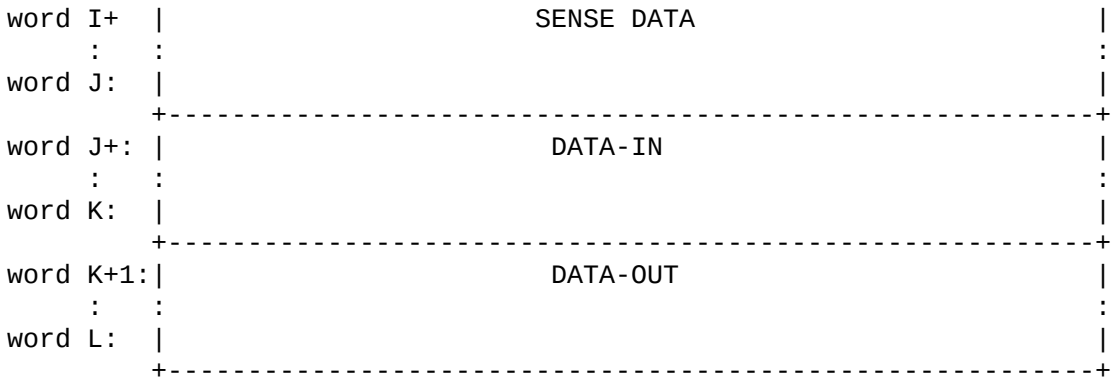
This command is used by a virtual disk client to set the geometry information for the disk volume the client is attached to. The supplied data structure has the same format as the get disk geometry command (VD_OP_GET_DISKGEOM).

29.2.3.11. VDisk SCSI Command (VD_OP_SCSICMD)

This command is used to deliver a SCSI packet to the vDisk server. It is implementation specific as to whether the server passes the received packet directly to a SCSI drive or whether it chooses to simulate the SCSI protocol itself. A server must not advertise this command if it does not support either capability.

The LDC cookie in the descriptor ring should point to the following data structure which describes the command arguments. The same buffer is also used to return the result of the command to the vDisk client.





The *cstat* field reports to the vDisk client the SCSI command completion status. SCSI command completion status are described in the SCSI Architecture Model documents[scsi3].

The *sstat* field reports to the vDisk client the SCSI command completion status of the SCSI sense request. SCSI command completion status are described in the SCSI Architecture Model documents[scsi3].

The *sstat* field is defined only if a SCSI sense buffer was provided and if the SCSI command completion status indicates that sense data should be available.

The *tattr* field defines the task attribute of the SCSI command to execute. The possible attributes are:

0x00	no task attribute defined
0x01	SIMPLE
0x02	ORDERED
0x03	HEAD OF QUEUE
0x04	ACA

Task attributes are defined in the SCSI Architecture Model documents[scsi3]. The vDisk server may ignore the task attribute.

The *tprio* field is a 4-bit value defining the task priority assigned to the SCSI command to execute. The task priority is defined in the SCSI Architecture Model documents[scsi3]. The vDisk server may ignore the task priority.

The *crn* field is a command reference number (CRN). SCSI command reference numbers are defined in the SCSI Architecture Model documents[scsi3]. The vDisk server may ignore the CRN.

The *reserved* field is reserved and should not be used.

The *timeout* field is the time in seconds that the vDisk server should allow for the completion of the command. If it is set to 0 then no timeout is required.

The *options* field is a bitmask specifying options for the SCSI command to execute. The possible bitmask values are:

0x01 (CRN)	This bitmask indicates that a command reference number (CRN) is specified in therequest.
0x02 (NORETRY)	This bitmask indicates that the vDisk server should not attempt any retry or other recovery mechanisms if the SCSI command terminates abnormally in any way.

The Command Descriptor Block (CDB) *length* field is set by the vDisk client and indicates the number of bytes available in the CDB field.

The *sense length* field is initially set by the vDisk client and indicates the number of bytes available in the sense field for storing sense data for SCSI commands returning with a SCSI command completion status indicating that sense data should be available. After the execution of the SCSI command, the vDisk server sets the *sense length* field to the number of bytes effectively returned in the *sense* field, or 0 if no sense data were returned.

The *data-in size* field is initially set by the vDisk client and indicates the number of bytes available for data transfers to the *data-in* field. After the execution of the SCSI command, the vDisk server sets the *data-in size* field to the number of bytes effectively transfered to the *data-in* field, or 0 if no data were transfered.

The *data-out size* field is initially set by the vDisk client and indicates the number of bytes available for data transfers from the *data-out* field. After the execution of the SCSI command, the vDisk server sets the *data-out size* field to the number of bytes effectively transfered from the *data-out* field, or 0 if no data were transfered.

The *CDB data* field contains the SCSI Command Descriptor Block (CDB) which defines the SCSI operation to be performed by the vDisk server. The structure of the CDB is part of the SCSI Standard Architecture[scsi3]. The size of the *CDB data* field should be equal to the number of bytes indicated by the vDisk client in the *CDB length* field rounded up to a multiple of 8 bytes.

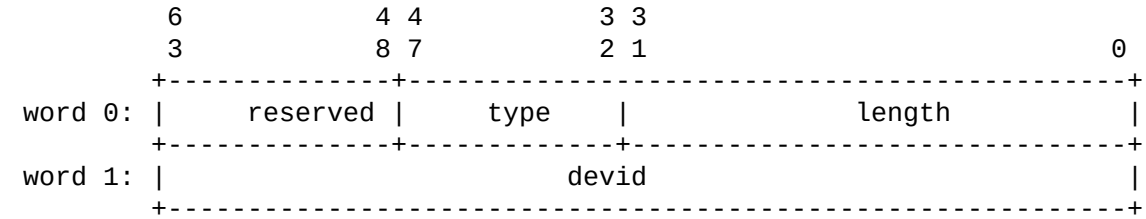
The *sense data* field contains sense data for SCSI commands returning with a SCSI command completion status indicating that sense data should be available. The structure of sense data is described in the SCSI Primary Commands documents[scsi3]. The size of the *sense data* field should be equal to the number of bytes indicated by the vDisk client in the *sense length* field rounded up to a multiple of 8 bytes.

The *data-in* field contains command specific information returned by the vDisk server at the time of command completion. The validity of the returned data depends on the SCSI command completion status. The size of the *data-in* field should equal to the number of bytes indicated by the vDisk client in the *data-in size* field rounded up to a multiple of 8 bytes.

The *data-out* field contains command specific information to be sent to the vDisk server. The size of the *data-out* field should be equal to the number of bytes indicated by the vDisk client in the *data-out size* field rounded up to a multiple of 8 bytes.

29.2.3.12. VDisk Get Device ID (VD_OP_GET_DEVID)

Device IDs[diskids] are persistent unique identifiers for devices in Solaris, and provide a means for identifying a device, independent of device's current name or instance number. This command is used to return the device ID of a disk volume backing a virtual disk. A successful completion of this command will result in the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.



```
6
```

The field *devid* contains the ID of the disk volume. The field *length* in the request should be set to the size of the buffer allocated by the vdisk client for storing the device ID. The vdisk server will then set it to the size of the returned devid in its response. The returned device ID value will be truncated if the provided space is not large enough to store complete ID. The field *type* specifies the type of device ID. Please refer to PSARC cases 1995/352 [http://arc.opensolaris.org/caselog/PSARC/1995/352/], 2001/559 [http://arc.opensolaris.org/caselog/PSARC/2001/559/], and 2004/504 [http://arc.opensolaris.org/caselog/PSARC/2004/504/], for a description of device IDs along and a list of the device ID type values.

29.2.3.13. VDisk Get EFI Data (VD_OP_GET_EFI)

This command is used to get EFI data for the disk volume a client is attached to. A successful completion of this command will result in the following data structure with the EFI data in the data field being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring. The returned data structure has the following format:

```
           3                                                           0
          +-------------------------------------------------------------+
word 0:   |                            LBA                              |
          +-------------------------------------------------------------+
word 1:   |                          length                             |
          +-------------------------------------------------------------+
word 2-N: |                         EFI data                            |
          +-------------------------------------------------------------+
```

The field *LBA* is the logical block address of the disk volume to get EFI data. Data returned in the EFI data field is determined by the value specified in the LBA field:

- If *LBA* is equal to 1, then the vdisk server should return the GUID Partition Table Header (GPT).
- If *LBA* is equal to the *PartitionEntryLBA* field from the GUID Partition Table Header, then the vdisk server should return the GUID Partition Entry array (aka GPE).

If the EFI data buffer is not large enough to return the request data then the vdisk server should return an error. The field *length* is the maximum number of bytes that can be stored in the data field of the provided structure.

The format of the GUID Partition Table Header and GUID Partition Entry are beyond the scope of this document and are defined in the Extensible Firmware Interface Specification[efi].

29.2.3.14. VDisk Set EFI Data (VD_OP_SET_EFI)

This command is used by a virtual disk client to set EFI data for the disk volume the client is attached to. The supplied data structure has the same format as for the get EFI command (VD_OP_GET_EFI).

The value of the LBA field determines the content of the EFI data field and the action taken by the vdisk server.

- If LBA = 1, then the vdisk server should use the contents of the EFI data field to set the GUID Partition Table Header (aka GPT).

- If LBA is equal to the PartitionEntryLBA field from the GUID Partition Table Header, then the vdisk server should the contents of the EFI data field to set the GUID Partition Entry array (aka GPE).

The format of the GUID Partition Table Header and GUID Partition Entry are beyond the scope of this document and are defined in the Extensible Firmware Interface Specification[efi].

29.2.3.15. VDisk Reset (VD_OP_RESET)

This command is used by the vDisk client to request the vDisk server to reset the disk or device being exported by it. It is implementation independent as to whether the server physically resets the underlying device or it chooses to only simulate a device reset.

Following a reset, any exclusive access rights or options that might have been set using the VD_OP_SET_ACCESS operation should be cleared in a way similar to receiving a VD_OP_SET_ACCESS operation with the CLEAR option.

In the event of a connection loss between the vDisk client and server, the vDisk server should behave as if it has received a VD_OP_RESET operation. It should clear any exclusive access rights or options set using the VD_OP_SET_ACCESS operation. A vDisk server implementing the disk reset is required to complete the operation prior to reestablishing the connection with the vDisk client.

29.2.3.16. VDisk Get Access (VD_OP_GET_ACCESS)

This command is used by the vDisk client to query whether it has access to the disk being exported by the vDisk server. The response has a payload of a single 64-bit unsigned integer, and may contain the following values:

0x00 (DENIED) The access to the disk is not allowed.

0x01 (ALLOWED) The access to the disk is allowed.

29.2.3.17. VDisk Set Access (VD_OP_SET_ACCESS)

This command is used by the vDisk client to request exclusive access to the disk being exported by the vDisk server. The payload is a single 64-bit unsigned integer. It can either contain a value of 0, or a bitmask of the following non-zero values:

0x00 (CLEAR) The vDisk server should clear any exclusive access rights, and restore non-exclusive, non-preserved access rights. In particular, the vDisk server should relinquish any exclusive access rights that have been acquired with the EXCLUSIVE flag, and disable any mechanism to preserve exclusive access rights enabled with the PRESERVE flag.

0x01 (EXCLUSIVE) The vDisk server should acquire exclusive access rights to the disk. When the vDisk server has exclusive access rights to the disk then any access to the disk from another host should fail. If another host already has acquired exclusive access rights to the disk then the vDisk server should fail to acquire exclusive access rights.

0x02 (PREEMPT) The vDisk server can forcefully acquire exclusive access rights to the disk. If another host has already acquired exclusive access rights to the disk, then the vDisk server can preempt the other host and acquire exclusive access rights.

0x04 (PRESERVE) The vDisk server should try to preserve exclusive access rights to the disk. The vDisk server should try to restore exclusive access rights if exclusive access rights are broken via random events (for example disk resets). When restoring

the exclusive access rights, the vDisk server should not preempt any other host having exclusive access rights to the disk.

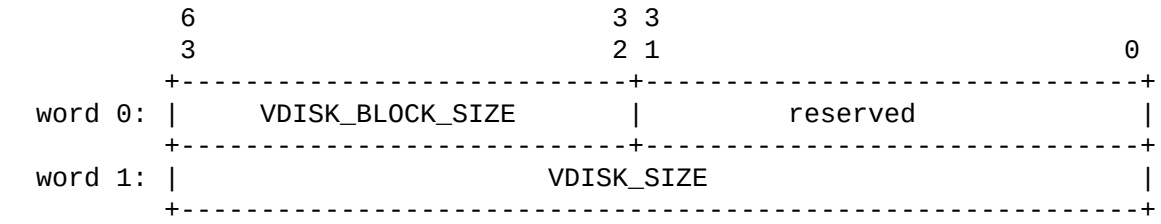
The PREEMPT and PRESERVE flags are only valid when the EXCLUSIVE flag is set.

In the event of a connection loss between the vDisk client and server, the vDisk server should perform the equivalent operation to a vDisk Reset Command (`VD_OP_RESET`) received from the client, and exclusive access rights and options should be cleared.

If the vDisk client still requires exclusive access rights following a connection reset, then it should send a new `VD_OP_SET_ACCESS` operation to the vDisk server and request exclusive access.

29.2.3.18. VDisk Get Capacity (`VD_OP_GET_CAPACITY`)

This command is used to get information about the capacity of the disk volume export by the vDisk server. A successful completion of this command will result in the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring:



The *`vdisk_block_size`* field contains the length in byte of the logical block of the vDisk. The *`vdisk_block_size`* should be the same value as the *`vdisk_block_size`* returned during the initial handshake as part of the attribute exchange.

The *`vdisk_size`* field contains the size of the vDisk in blocks specified as a multiple of *`vdisk_block_size`*.

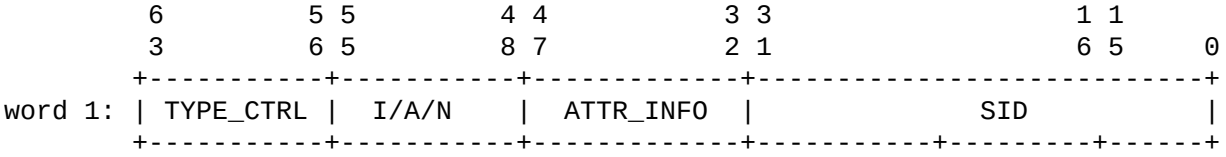
If the vDisk server is unable to obtain the vDisk size, it should set the *`vdisk_size`* to `-1`. Under these circumtances, the vDisk client can retry the operation later to check if the size is available.

29.3. Virtual network protocol

This section describes the packet formats and protocol used for the virtual networking infrastructure between logical domains.

29.3.1. Attribute information

During the initial handshake, as part of the `CTRL/INFO/ATTR_INFO` message, the virtual network device will exchange information with the virtual switch and other vNetwork devices about the transfer protocol, its address and MTU. The format of the attribute payload is shown below:



```
word 2: | XFER_MODE | ADDR_TYPE |  ACK_FREQ   | PLNK_UPDT | OPTIONS | rsvd |
        +-----------+-----------+-------------+-----------+---------+------+
word 3: |                              ADDR                                 |
        +-------------------------------------------------------------------+
word 4: |                               MTU                                 |
        +-------------------------------------------------------------------+
```

The sending client, be it a virtual network device and/or virtual switch will provide its peer with the transfer mode, acknowledgment frequency, address, address type and MTU it intends to use for sending network packets. The peer ACKs the attribute message if it agrees to all the parameters.

Currently the only supported address type is:

VNET_ADDR_ETHERMAC	1	Ethernet MAC Address

The *addr* field contains the mac address of the client sending the attribute information.

If VIO version 1.3 or lower is negotiated, it is required that the MTU exchanged by either ends during the attribute exchange matches exactly. If version 1.4 or higher is negotiated, and the MTU received in the ATTR/INFO doesn't match the receiver's MTU, it ACKs with the lower of the two MTUs. All subsequent communication between both ends are required to use the mutually agreed upon MTU.

If VIO version 1.4 or lower is negotiated, bits 32-63 in word-2 are reserved; i.e., they must be set to 0 and will be ignored by the peer. If VIO version 1.5 is negotiated, the *PLNK_UPDT* field (bits 32-39) is used to indicate any physical link information updates that a vNet device is interested in. Bits 40-63 are reserved. A vNet device could negotiate with the vSwitch device to obtain updates about certain physical link properties. Only “physical link status” updates are supported for now and only the lower 2 bits of this 8-bit field are defined and the remaining bits within this field are reserved.

A vNet device that desires to get physical link status updates sets this field to the appropriate value (see bit definitions below) in its ATTR/INFO message to the vSwitch. Depending on its capabilities, the vSwitch device either ack's or nack's by updating these bits in its response message. Note that a vSwitch device must not nack the attribute message itself simply because it cannot support link status notifications; the physical link update bits only indicate the desire by the vNet device and it is not guaranteed that the vSwitch device will be able to provide that information. Thus, if the rest of the information in the ATTR/INFO message is acceptable to the vSwitch except *PLNK_UPDT* bits, then only the *PLNK_UPDT* field must be nack'd by setting the appropriate bits; and the attribute message itself should be acknowledged by sending a ATTR/ACK message. Also, note that these bits are relevant only when the peers involved in the attribute exchange are a vNet device and a vSwitch. The bits are reserved and must be ignored during handshake between two vNet peers.

Bit definitions of the *PLNK_UPDT* field:

PHYSLINK_UPDATE_NONE	0	No plink props desired
PHYSLINK_UPDATE_STATE	1	Need plink state updates
PHYSLINK_UPDATE_STATE_ACK	2	Can update plink state
PHYSLINK_UPDATE_STATE_NACK	3	Cannot update plink state

For further information on the protocol to communicate physical link updates, refer to Section 29.3.5, “Network Device Physical Link Information Updates”

Starting with version 1.6 of the VIO protocol, the virtual network and virtual switch devices support descriptor rings in VIO_RX_DRING_DATA mode, in addition to the modes that are supported in earlier

versions of the protocol. If version 1.6 is negotiated, the *OPTIONS* field (bits 40-47) is used to indicate the specific descriptor ring mode(s) the VIO device wants to operate in. The supported values for the options in version 1.6 of the VIO protocol are:

VIO_TX_DRING	1	Transmit descriptor ring
VIO_RX_DRING	2	Receive descriptor ring
VIO_RX_DRING_DATA	4	Receive descriptor ring with data area

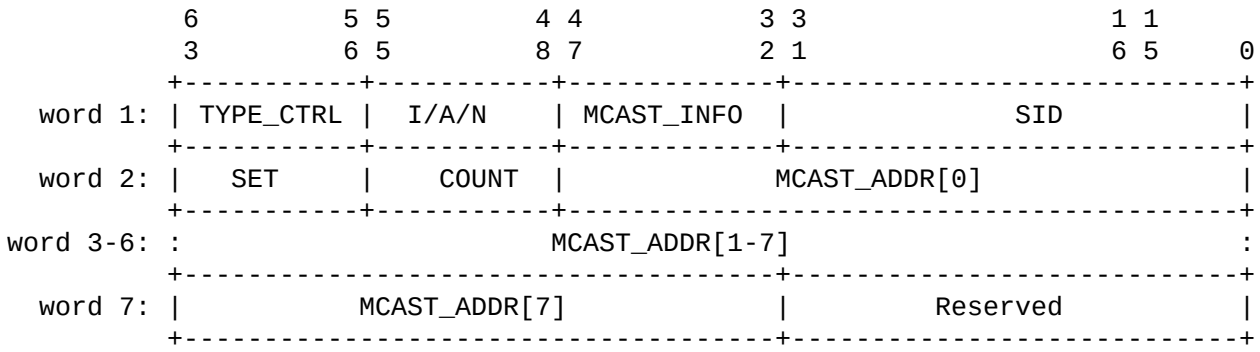
A VIO device and its peer negotiate the specific dring mode in which they will communicate with each other, as part of their attribute negotiation. Though the version negotiated is 1.6, a device and/or its peer can choose not to operate in RX_DRING_DATA mode. Also, a device can choose to operate in RX_DRING_DATA mode with only some of its peers. A device must indicate the specific dring mode(s) that it can negotiate with its peer, by setting the corresponding bits in the options field. The peer reads this field. If at least one of the modes is acceptable, it responds by sending an ACK message. In its ACK message, it leaves only the bit corresponding to the mode it chooses and clears the remaining bits, even if more than one mode is acceptable. If the peer does not support any of the modes requested in the message, it responds by sending a NACK message.

29.3.1.1. Multicast information

Virtual network devices can set/unset the multicast groups they are interested in to a virtual network switch at any point after a succesful handshake and during normal data transfer. Each packet sent by a vnet device is of type CTRL/INFO/MCAST_INFO.

VNET_MCAST_INFO	0x101	Multicast information

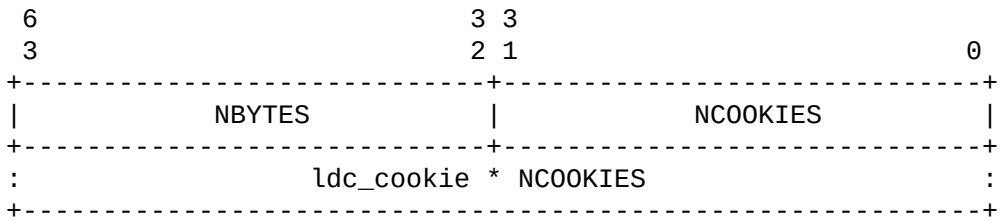
If the *set* field is equal to '1', then the corresponding mcast addresses are being set by the vnet device, or else the switch assumes that the specified address(es) are being removed. The peer will ACK the info packet if it successfully registered or removed the specified multicast mac addresses. If the multicast address was already set earlier or if the network device tries to unset an address that was not set earlier, the virtual switch will NACK the request. The MCAST_ADDR field can contain a max of VNET_NUM_MCAST (7) multicast addresses, where each address is ETHERADDRL (6) bytes in length. The count field specifies the actual number of multicast addresses in the packet.



29.3.2. vNet descriptors

The virtual network and virtual switch devices that use hypervisor shared memory will send and receive Ethernet frames by specifying the various fields in each descriptor. In VIO_TX_DRING mode, the de-

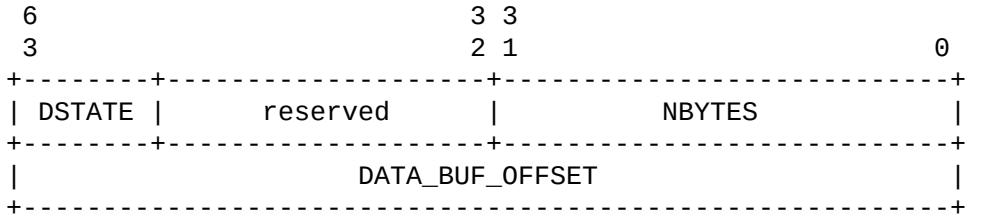
scriptor format consists of a header that is common to VIO clients of all classes (See Section 29.1.4.2.3, “Descriptor Ring Data Message Format (common to all dring modes)”) and a class specific part that is defined by the specific device class. The format of the class specific descriptor is shown below.



In this format, the peers send and receive Ethernet frames by specifying the length of data and the LDC memory cookies corresponding to the pages containing the frame in each descriptor. See Section 29.1.3.3, “Descriptor ring registration” for more information about the LDC transport cookie.

The *nbytes* field specifies the number of bytes being transmitted. The *ncookies* and *ldc_cookie* fields refer to the segment of memory from/to which data is being read/written.

In VIO_RX_DRING_DATA mode, the descriptor consists of only the device class specific part, with no common header part, as shown below.



DSTATE This field specifies the state of the descriptor. The valid state values and usage are same as those described in the case of common descriptor header in Section 29.1.4.2.3, “Descriptor Ring Data Message Format (common to all dring modes)”.

NBYTES The size of the ethernet frame in the data buffer. This field is set by the VIO device that is transmitting the frame.

DATA_BUF_OFFSET The VIO device which is exporting the descriptor ring and its associated data buffers sets this field in each descriptor. The field is set to the offset of the data buffer within the data buffer area, that is assigned to this descriptor. The importing device must copy the frame to be transmitted to the buffer corresponding at this offset.

Initially during descriptor ring registration, every descriptor must be initialized by the exporting VIO device. The *DATA_BUF_OFFSET* should be set to the offset of the specific buffer in the data buffer area that is assigned to the descriptor. The descriptor state must be set to VIO_DESC_FREE. When the peer VIO device (importing end point) needs to transmit a frame, it determines the buffer based on the buffer offset specified in the descriptor and will copy the frame to be transmitted to this address. It will mark the *NBYTES* field to reflect the size of the frame being transmitted. It will mark the *DSTATE* field as VIO_DESC_READY. It will then send a DRING_DATA message if necessary as described in Sec-

tion 29.1.4.2.3, “Descriptor Ring Data Message Format (common to all dring modes)”. The receiving VIO device will process the corresponding descriptor and its associated buffer. After processing the descriptor, the receiver may specify a new data buffer offset value (note this is not necessary and implementation specific) or keep the existing offset, before marking the *DSTATE* as VIO_DESC_DONE. It then continues to process the next descriptor and will finally send a DRING_DATA ack message with a *proc_state* value of VIO_DP_STOPPED, to the transmitting peer. The transmitter must always read the data buffer offset field in the decriptor every time it needs to transmit a frame, after verifying that the *DSTATE* is VIO_DESC_DONE. The transmitting VIO device must not assume that that data buffer offset remains the same.

29.3.3. Virtual LAN (VLAN) support

The VIO protocol for virtual network and switch devices will be extended in version 1.3 to include support for virtual LANs (VLANs) as specified by the IEEE 802.1Q4 specification. A VLAN aware network or switch device will be capable of sending, receiving or switching Ethernet frames that contain a VLAN tagged header. If a network/switch device negotiates version 1.3 or higher with its peer, the MTU size it specifies in the attribute info message (Section 29.3.1, “Attribute information”) should correspond to the size of a tagged Ethernet frame. Similarly, if a peer negotiates version 1.2 or lower, sending/receiving tagged frames can result in undefined behavior including the frames being dropped.

29.3.4. Network Device Resource Sharing via DDS

The VIO DDS control message provides the capability to share device resources between VIO device peers. The DDS framework will be primarily used by a vSwitch device to share the underlying physical network device's resources with a vNet device.

All DDS messages for vNet and vSwitch devices will contain a class field that uniquely identifies the type of device from which the resources are being shared. In version 1.3 of the VIO protocol, the vNet device will define a new DDS message class DDS_VNET_NIU for sharing the resources of a UltraSPARC-T2 NIU device.

DDS_VNET_NIU	0x10	NIU vNet class

Each DDS message of class VNET_NIU sent by a vSwitch or a vNet will contain a subclass field that specifies the requested operation. The DDS subclass values for a VNET_NIU class are:

DDS_VNET_ADD_SHARE	1	Add a device share
DDS_VNET_DEL_SHARE	2	Remove a device share
DDS_VNET_REL_SHARE	3	Release a device share
DDS_VNET_MOD_SHARE	4	Modify a device share

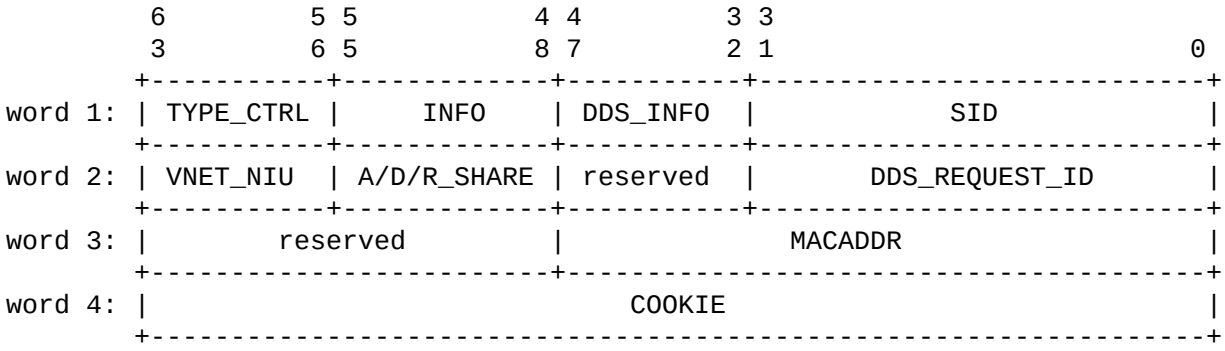
The DDS_VNET_(ADD/DEL/REL)_SHARE messages subclasses are used when adding or deleting a resource to a domain or releasing a resource from a domain.

The ADD_SHARE message is used by the vSwitch device to add a virtual region resource uniquely identified by its cookie to a vNet device identified by its macaddr.

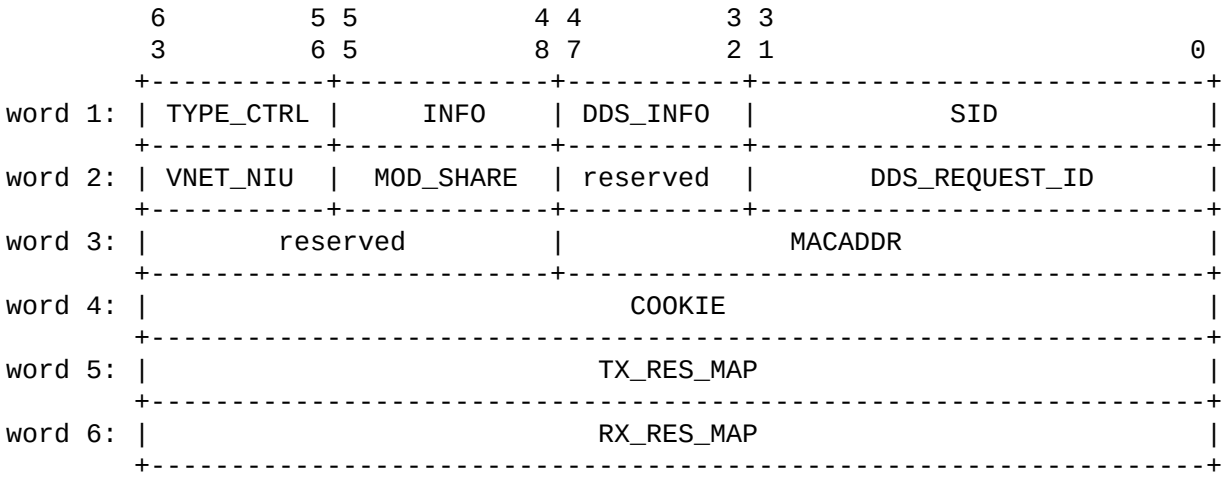
The DEL_SHARE message is similarly used by the vSwitch to remove a virtual region resource that was previously added using the ADD_SHARE operation.

The REL_SHARE message is used by the vNet device to inform the vSwitch device that it is no longer using a previously added shared resource. The vSwitch on receiving a REL_SHARE message can reclaim and reassign the resource to another vNet. A vNet device should not attempt to use a resource that it had

previously released via the REL_SHARE operation. The message format for the add, delete and release operations is identical and is shown below:

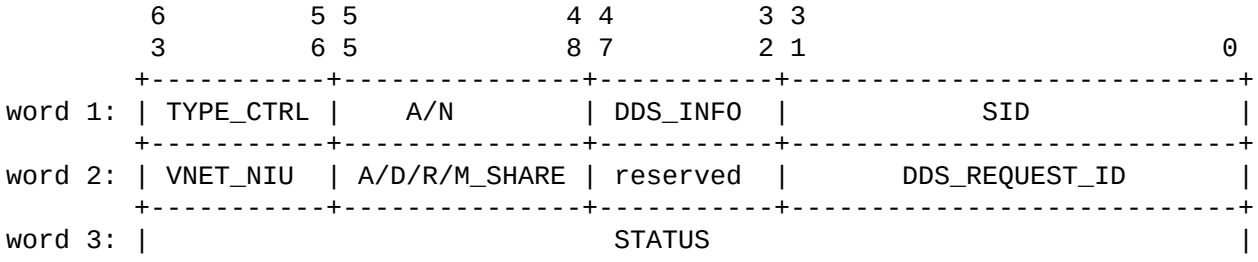


The resource modification operation allows a vSwitch device to modify the contents of a shared virtual region. In addition to the *macaddr* and *cookie* fields, the message also contains a updated map of TX and RX resources assigned to the virtual region resource. The format of the modify message is shown below:



In addition to the different CTRL/INFO/DDS_INFO request messages, the vNet and vSwitch devices will also ACK and NACK all received DDS requests. The ACK and NACK responses will contain a *status* field that specify the outcome of the requested operation.

The format of the ACK/NACK response message is below:



```
        +-------------------------------------------------------------+
```

The currently defined ACK and NACK status values are:

DDS_VNET_SUCCESS	0	Operation was successful
DDS_VNET_FAIL	1	Operation failed

29.3.5. Network Device Physical Link Information Updates

The VIO protocol for virtual network and virtual switch devices was extended in version 1.5 to include support for physical link property updates. A vNet device will be able to negotiate for physical link updates, as part of its attribute exchange phase of handshake with the vSwitch. Currently, physical link state is the only property that can be negotiated for updates. See Section 29.3.1, “Attribute information” for details on the attribute message.

Once a vNet device successfully negotiates physical link state updates, the vSwitch must send an initial update about the physical link status right after the handshake is complete. Further, whenever the physical link status changes, the vSwitch must keep updating it to the vNet device, until either the connection is terminated by the vNet device or the Channel goes down or gets reset.

The packet sent by the vSwitch device to a vNet device is of type CTRL/INFO/VNET_PHYSLINK_INFO. The bits within the *physlink_info* field indicate the physical link property and its current value that is being updated to the vNet device. Currently, the lower 2 bits are defined to indicate the physical link state and the remaining bits are reserved. The vNet device on receiving this should send a message of type CTRL/ACK/VNET_PHYSLINK_INFO back to the vSwitch. The vNet device can choose to either ignore or nack the message, if it has not negotiated with the vSwitch for physical link updates or if the message is received while handshake with the vSwitch device is still in progress.

VNET_PHYSLINK_INFO	0x103	Physical Link Information

The format of the physical link information message is as shown below:

```
         6         5 5          4 4          3 3                          0
         3         6 5          8 7          2 1                          |
        +-----------+------------+------------+--------------------------+
word 1: | TYPE_CTRL |   I/A/N    | PLINK_INFO |           SID            |
        +-----------+------------+------------+--------------------------+
word 2: |          reserved                    |      physlink_info       |
        +--------------------------------------+--------------------------+
word 3: |                            reserved                             |
        +-----------------------------------------------------------------+
word 4: |                            reserved                             |
        +-----------------------------------------------------------------+
```

Bit definitions of the *physlink_info* field:

VNET_PHYSLINK_STATE_DOWN	1	Physical Link State: Down
VNET_PHYSLINK_STATE_UP	2	Physical Link State: Up
VNET_PHYSLINK_STATE_UNKNOWN	3	Physical Link State: Unknown

Chapter 30. Domain services

30.1. Overview

In a Logical Domain environment the ability to discover whether a guest operating system has various capabilities, and be able to remotely direct it to perform various operations is important. Similarly it is equally important for a guest operating system to be able to discover and communicate with its various support services.

Specifically, each guest domain can offer a number of capabilities to its service entity, and similarly the service entity can offer a set of capabilities for use by the guest domain.

Capabilities may include things such as the ability to perform dynamic reconfiguration, or be directed to perform a graceful shutdown or reboot by a service entity.

As a domain transitions through various operational phases, (for example while booting) its capabilities may change. The capabilities of a simple guest OS like OpenBoot are not the same as those of a full blown operating system such as Linux or Solaris. Similarly services that are offered to a domain by its service entity/entities may come and go if, for example, a service processor re-boot occurs.

Consequently it is a requirement that the mechanism for capability discovery and communication must be able to cope with the dynamic nature of both a guest domain and its service entities.

This section describes the protocol by which a guest OS may register its capabilities with its service entity/entities, and vice-versa. The registration process includes independent version negotiation between client and service for each capability.

Once a capability has been registered, the domain services protocol then provides a data transport for client and service to communicate directly with each other independently of other capability services which may be using the same channel.

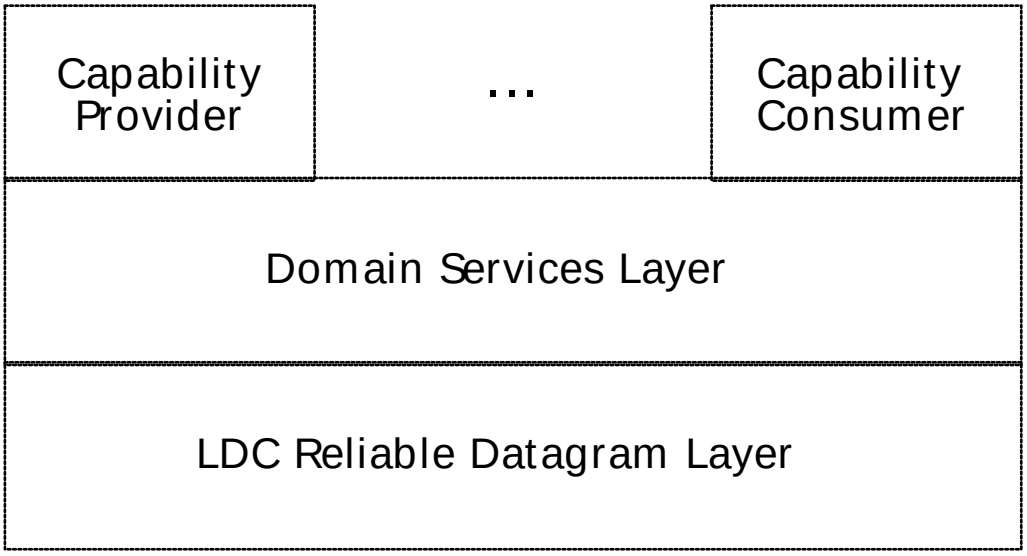
30.1.1. Communication Stack

The domain services (DS) mechanism is layered on top of domain channels to facilitate communication between a guest domain and its service entities. The reliable mode protocol of the Logical Domain Channel (LDC) framework is leveraged to ensure in-order guaranteed packet delivery as well as detection of faults on the communication channel— including loss of connection due to, say, the communication peer crashing or re-booting.

On top of the LDC reliable protocol the DS protocol handles the registration of provider capabilities with their consumer(s), and subsequently the routing of data messages for those registered capabilities.

The content of transported messages is specific to the higher-level protocol between the particular DS service and its client. The DS communication stack is illustrated below.

Figure 30.1. Domain Service communication stack Layers

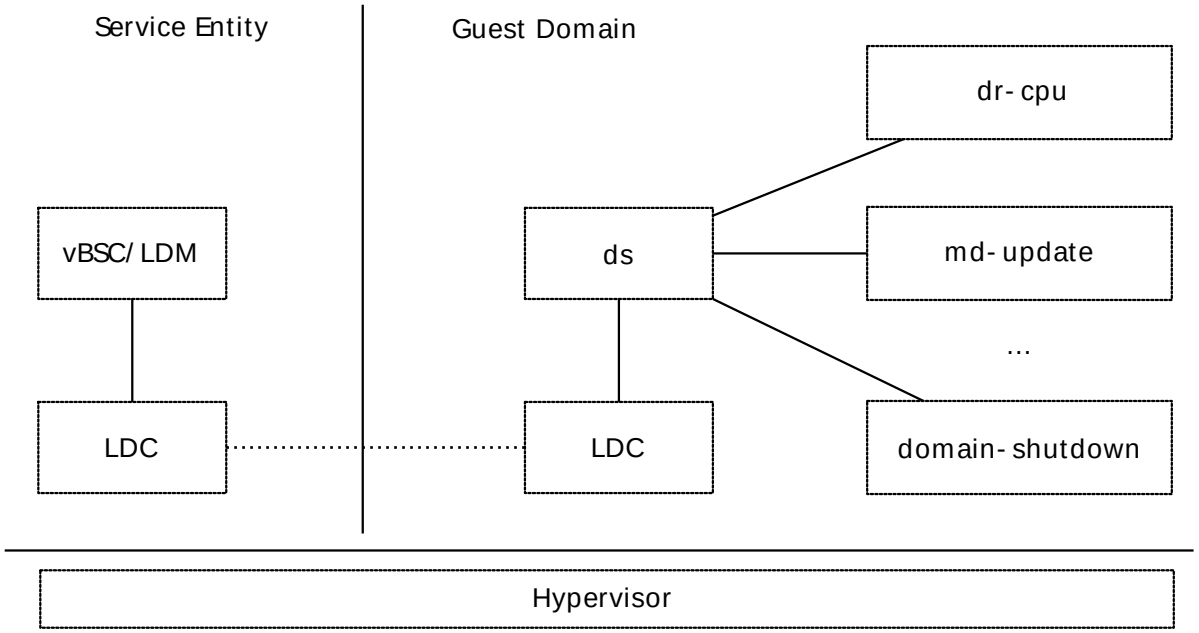


By analogy, just as LDC provides a low level transport, like IP, the domain services protocol provides a name service and connection transport protocol, like TCP, to facilitate communication between a capability provider and its consumer.

Messages for a set of registered capabilities are multiplexed over a shared LDC channel.

This basic communication flow is illustrated below.

Figure 30.2. Domain Services Communication Path Example



30.2. Domain Services Protocol

30.2.1. Definitions

Unless otherwise stated, each of the fields and sizes specified herein are given in bytes (octets). Byte ordering for multi-byte fields is network byte order (big-endian). All variable-length character array definitions are assumed to be NUL-terminated sequences of ASCII values, with a maximum length (including the terminating NUL) less than or equal to the constant MAX_STR_LEN, defined as:

MAX_STR_LEN	1024

30.2.2. DS Message Header

All DS messages consist of a fixed sized header followed by a variable length data payload. The header format is as follows:

Offset	Size	Field name	Description
0	4	*msg_type*	Message type
4	4	*payload_len*	Payload length

The data payload content is defined according to the *msg_type* field.

30.2.3. DS protocol fixed message types

The DS protocol always supports three message types and payloads, as described below, independent of the current version of the protocol. The type-specific payload is described below each type.

The message types described in this section are intended for version negotiation of the basic DS protocol. All other message types are undefined until the DS protocol version has been negotiated.

The underlying LDC reliable protocol layer will ensure error-free packet delivery, so corrupted packets will already have been dropped. However, receipt of unknown packet types may still occur as a result of bugs or due to malicious guest OS behavior. Upon the receipt of an unknown or undefined (for the currently negotiated DS protocol version) packet type, the recipient should discard the datagram, and close the LDC channel. This action resets the domain services channel connection. Re-opening the channel again should ensure complete end-to-end protocol negotiation and re-registration of capabilities.

30.2.4. Initiate DS connection

msg_type:

DS_INIT_REQ	0x0

Payload:

Offset	Size	Field name	Description
0	2	*major_vers*	Requested major number
2	2	*minor_vers*	Requested minor number

30.2.5. Initiation acknowledgment

msg_type:

`DS_INIT_ACK` `0x1`

Payload:

Offset	Size	Field name	Description
0	2	*minor_vers*	Highest supported minor version

30.2.6. Initiation negative acknowledgment

msg_type:

`DS_INIT_NACK` `0x2`

Payload:

Offset	Size	Field name	Description
0	2	*major_vers*	Alternate supported major version

30.2.7. DS protocol version negotiation

The DS protocol negotiation involves a countdown algorithm in an attempt to agree on a common major number. Major numbers correspond to incompatible changes; both sides must agree on a major version number for the version negotiation to proceed. As part of agreeing on a major number agreement, each side learns of the other's highest supported corresponding minor number. Minor numbers correspond to backwards-compatible changes; the two sides implicitly agree to use the lower of the two minor numbers exchanged, and the negotiation is successfully completed.

Specifically, the negotiation is initiated by the guest sending the `DS_INIT_REQ` message to the service entity listening on the other end of the domain channel. This message includes major and minor version numbers supported by the guest.

If the service entity can't support the major version number sent from the guest, it responds with the `DS_INIT_NACK` message, specifying the closest major version number it can support. The guest can then initiate a new negotiation if it wants (i.e. if it can support the alternate major number returned by the service entity). However, if the service entity's `DS_INIT_NACK` message includes a major number of zero, the service entity should assume that the guest does not support any version of the DS protocol in common with it.

If the major number sent in the `DS_INIT_REQ` message is one the service entity supports, it returns a `DS_INIT_ACK` message specifying the highest minor number of the protocol version it supports. Since minor number changes correspond to compatible protocol changes, once the guest receives the `DS_INIT_ACK` message, both sides can communicate using the version of the protocol corresponding to the major number agreed to, and the lower of the two minor numbers exchanged The version negotiation is now successfully completed.

30.3. DS protocol version 1.0

30.3.1. Service Handles

A service handle (`svc_handle`) is an opaque 64-bit descriptor that uniquely identifies an instance of a service. It is analogous to a TCP port number, and is specified as part of the `DS_REG_REQ` message (described in Section 30.3.4.1, “Register Service”), sent to begin the negotiation/registration process for a capability. It is used during this phase to identify the specific negotiation in progress (there could be more

than one). Once a capability has been registered, it is used to identify the entity to be notified on receipt of a message. Similarly, when a capability sends a message to a client, the handle identifies the sender. It also identifies the target service during the unregistration process.

30.3.2. Service Identifier

The DS_REG_REQ message also specifies a service identifier (*svc_id*), a NUL-terminated character string naming the service. The format and restrictions on the *svc_id* string are identical to the PROP_STR type's data field as defined in Chapter 8, *Machine description*.

30.3.3. Result Codes

Some of the response message types defined herein include a result field in their payload to indicate a reason for failure. The complete list of such failure codes is presented here. The definition of each is included in the section describing the response message type to which it belongs.

DS_REG_VER_NACK	0x1
DS_REG_DUP	0x2
DS_INV_HDL	0x3
DS_TYPE_UNKNOWN	0x4

30.3.4. DS Message types defined for v.1.0 of the DS protocol

30.3.4.1. Register Service

msg_type:

DS_REG_REQ	0x3

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle
8	2	*major_vers*	Requested major version
10	2	*minor_vers*	Requested minor version
12	*variable length*	*svc_id*	Service name

30.3.4.2. Register Acknowledgment

msg_type:

DS_REG_ACK	0x4

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle sent in DS_REG_REQ
8	2	*minor_vers*	Highest supported minor version

30.3.4.3. Register Failed

msg_type:

DS_REG_NACK 0x5

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle sent in DS_REG_REQ
8	8	*result*	Reason for the failure
16	2	*major_vers*	Alternate supported major version

A DS_REG_NACK message can return the following result codes:

DS_REG_VER_NACK Cannot support requested major version

DS_REG_DUP Duplicate registration attempted

30.3.4.4. Unregister Service

msg_type:

DS_UNREG 0x6

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle to unregister

30.3.4.5. Unregister OK

msg_type:

DS_UNREG_ACK 0x7

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle sent in DS_UNREG

30.3.4.6. Unregister Failed

msg_type:

DS_UNREG_NACK 0x8

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle sent in DS_UNREG

30.3.4.7. Data Message

msg_type:

DS_DATA 0x9

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle that is the destination of the data message

Note

The DS_DATA header is defined so that when combined with the basic DS header the final payload delivered to the service is aligned on a 64-bit boundary with regard to the entire DS datagram delivered by LDC.

This alignment is to enable an implementation to potentially utilize an optimized copy when/if creating a message buffer for the final destination service.

30.3.4.8. Data Error

msg_type:

DS_NACK 0xa

Payload:

Offset	Size	Field name	Description
0	8	*svc_handle*	Service handle sent in DS_DATA
8	8	*result*	Reason for failure

A DS_NACK message can return the following result codes:

DS_INV_HDL Service handle not valid
DS_TYPE_UNKNOWN Unknown *msg_type* received

30.3.5. DS Capability Version Negotiation & Registration

Version negotiation for DS capabilities utilizes exactly the same countdown algorithm as used in the DS Protocol version negotiation, with the same semantics for major & minor numbers, and corresponding message types for implementation. The details of that portion of the protocol are not repeated here.

The registration process is the way in which DS capabilities advertise their availability. A registration is initiated by the service sending a DS_REG_REQ message containing both a service handle and a service identifier.

In response to a successful registration, the other side sends back a DS_REG_ACK message that includes the same service handle provided in the original message. Until this response is received, the DS service interface for this client is not available.

A DS_REG_NACK message is returned if the protocol major version numbers do not match (result: DS_REG_VER_NACK) or if a service with the same service ID is already registered (result: DS_REG_DUP).

This negotiation/registration handshake must occur whenever the underlying LDC comes up. If there is an event that causes the LDC to go down, all services are automatically unregistered. When the channel comes back up, all services must therefore re-register themselves.

30.3.6. Service Requests

Once the registration handshake has occurred, a DS client can send data messages to any of the registered servers by sending a DS_DATA message.

The data message payload includes the *svc_handle* of the service that is the intended recipient of the message. Following that is any service-specific payload. The *payload_len* field of the header is the length of the entire payload.

The final recipient of the message payload does not receive the DS header or the *svc_handle*. It only receives the remainder of the payload and an indication of the length of that portion of the payload.

If there is an error in the message that results in the inability of DS to forward the message to the intended recipient, a DS_NACK reply message is sent back with an error indication of either DS_INV_HDL (invalid *svc_handle*) or DS_TYPE_UNKNOWN (unknown *msg_type* received) in the result field. Note that the original payload is not returned.

If the message is forwarded all the way to the service successfully, the higher level protocol implemented by that service determines what if any reply message is sent.

30.3.7. Unregistration

In the event that a capability becomes unavailable, such as if the kernel module that provides it is unloaded, a DS_UNREG message is sent.

The *svc_handle* field of the DS header is filled with the service handle that uniquely identifies the registered service. There is no payload to this message.

Once the first message is received, the service handle is invalidated and connections to that service are closed.

If the DS LDC channel goes down, all registered services are forced to the unregistered state by one or both sides that are still running. Before a service can be used again, both the DS infrastructure handshake and the service registration handshake must be re-negotiated.

Service handles should not be reused after a service is unregistered. This prevents successful use of a stale handle. Service handles may be re-used after the basic LDC connection is taken down and then up, and the overall DS framework is reset as a result.

30.4. DS Capabilities

A DS capability is defined as any service provided by one subsystem on behalf of another. Capabilities are based on functionality rather than software module boundaries. Thus, a module can register multiple capabilities if it provides multiple features that are logically grouped together. Associated with a capability are a service identifier and a service handle.

The following sections describe the core DS capabilities supported in a Logical Domain environment.

30.5. MD Update Notification version 1.0

The MD update capability allows a service entity to notify a guest when the entity has modified the guest's Machine Description. It is the responsibility of the MD update capability to parse the new MD, determine what has changed, and initiate the steps required to adjust the guest configuration accordingly. The exact steps taken upon receiving an MD update notification may vary depending on the type of guest running in the domain.

30.5.1. Service ID

The following service ID should should be added to the Domain Services registry for the MD Update capability.

Service ID	Description
`"md-update"`	Notification of MD updates

30.5.2. MD Update Request

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number

The *req_num* field is used to match up request and response messages; the same number is used in the request and its associated response; the value itself is opaque to the clients of the protocol.

30.5.3. MD Update Response

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*result*	Result of operation

An MD update response message can return the following result codes:

```
MD_UPDATE_SUCCESS                0x0
MD_UPDATE_FAILURE                0x1
MD_UPDATE_INVALID_MSG            0x2
```

30.6. Domain Shutdown version 1.0

The Domain Shutdown capability allows a service entity to send a DS_DATA message requesting a guest to gracefully shutdown. The response indicates whether the request was successful (i.e. initiation of shutdown has occurred). If the request is denied, the response can include an informational message, encoded as a NUL-terminated ASCII string, describing the reason for denying the request (e.g. something like “DR in progress”).

30.6.1. Service ID

The following service ID should should be added to the Domain Services registry for the Domain Shutdown capability.

Service ID	Description
"domain-shutdown"	Request a graceful shutdown

30.6.2. Domain Shutdown Request

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*ms_delay*	milliseconds to delay

ms_delay specifies a time delay in milliseconds before initiation of the shutdown operation.

30.6.3. Domain Shutdown Response

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*result*	Result of operation
12	*variable*	*reason*	ASCII String (NUL-terminated)

reason is a NUL-terminated ASCII string.

A domain shutdown response message can return the following result codes:

```
DOMAIN_SHUTDOWN_SUCCESS          0x0
DOMAIN_SHUTDOWN_FAILURE          0x1
DOMAIN_SHUTDOWN_INVALID_MSG      0x2
```

30.7. Domain Panic version 1.0

The Domain Panic capability allows a service entity to send a DS_DATA message requesting a guest to panic and cause a crash dump to be created. The response indicates whether the request was successful (i.e. initiation of panic processing has occurred). If the request is denied, the response can include an informational message, encoded as a NUL-terminated ASCII string, describing the reason for denying the request (e.g. something like “DR in progress”).

30.7.1. Service ID

The following service ID should should be added to the Domain Services registry for the Domain Panic capability.

Service ID	Description
"domain-panic"	Request a panic

30.7.2. Domain Panic Request

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number

30.7.3. Domain Panic Response

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*result*	Result of operation
12	*variable*	*reason*	ASCII String (NUL-terminated)

reason is a NUL-terminated ASCII string.

A domain panic response message can return the following result codes:

```
DOMAIN_PANIC_SUCCESS                0x0
DOMAIN_PANIC_FAILURE                0x1
DOMAIN_PANIC_INVALID_MSG            0x2
```

30.8. CPU DR Version 1.0

The ability to add or remove virtual CPUs from a logical domain is driven from the LDom manager through this domain service.

30.8.1. Service ID

The following service ID should should be added to the Domain Services registry for the CPU DR capability.

Service ID	Description
"dr-cpu"	Dynamic reconfiguration for virtual CPUs

Each DR service message consists of a fixed message header and packet payload as described below. The overall payload length is determined by subtracting the size of the CPU DR message header (4 bytes) from the entire domain services packet size.

30.8.2. CPU DR Message Header

All CPU DR messages begin with the same header. The payload that follows the header is specific to a particular message type.

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*msg_type*	Message type
12	4	*num_records*	Number of records for message

The overall CPU DR protocol consists of a command sent to the client guest that then responds with a reply indicating the overall success of the request. An error response indicates that the operation was not

attempted due to an invalid request. A DR_CPU_OK response indicates that the requested operation was attempted and the response record for each cpu indicates the effect of the attempt for that particular cpu. The message types identifies either a request or a response to a request.

30.8.3. Message types

The following constants are defined for CPU DR domain service command identifier values.

Request message types:

Type	Value	ASCII Value	Definition
DR_CPU_CONFIGURE	0x43	'C'	Configure new CPU(s)
DR_CPU_UNCONFIGURE	0x55	'U'	Unconfigure CPU(s)
DR_CPU_FORCE_UNCONFIG	0x46	'F'	Forcibly unconfigure CPU(s)
DR_CPU_STATUS	0x53	'S'	Request the status of CPU(s)

Response message types:

Type	Value	ASCII Value	Definition
DR_CPU_OK	0x6f	'o'	Request completed successfully
DR_CPU_ERROR	0x65	'e'	Request failed (not attempted)

30.8.3.1. CPU DR Request records payload

The CPU DR requests all use the same message payload format, which is a list of records of virtual CPU IDs within a guest. The number of records of IDs is specified by the num_records field in the packet header. Each ID is given as a single 4 byte value.

The payload layout is as follows:

Offset	Size	Field name	Description
0	4	*id0*	First virtual CPU ID
4	4	*id1*	Second virtual CPU ID
8	4	*id2*	Third virtual CPU ID
...	4	*idN*	*N*th virtual CPU ID

Note: IDs should be provided in ascending numerical order, and should not be duplicated. An implementation may not assume that IDs are arranged in a specific order, and may not assume that IDs are not duplicated.

30.8.3.2. Request number

The request number in the message header is a monotonically increasing number that uniquely identifies each request message.

Responses to requests are expected to use the same request number so that they can be paired with their original request.

New requests may be issued without waiting for a response to a preceding request. The underlying transport protocol is responsible to ensure reliable, in-order and un-duplicated message packets.

Requests are to be processed in the order received.

30.8.3.3. DR_CPU_CONFIGURE request

This command requests that a guest providing this service attempt to configure and bring online a set of CPUs that have been dynamically reconfigured into the guest's logical domain.

The response to this request indicates success of failure for each individually specified CPU.

Before a configure request, a CPU must be part of the logical domain in the hypervisor and must be present in the guest's Machine Description. If either of these conditions is not satisfied, the configure response will indicate that the particular CPU is in the `DR_CPU_STAT_NOT_PRESENT` state. No other assumptions may be made about the state of the CPU before a configure request. In particular, attempts to configure a CPU already in the configured state must succeed.

If the guest provides a service for registering a Machine Description update, that update notification must be provided to the guest prior to the configure request being given.

After a successful configure request, a CPU is in the configured state, which means that it is available for general use by the guest. The CPU enters the guest from the hypervisor by means of the `cpu_start` hypervisor API (FWARC 2005/116 [http://arc.opensolaris.org/caselog/FWARC/2005/116]). Further steps required to reach the configured state is guest operating system specific. See [lddr] for details on the Solaris-specific implementation of the configure request.

30.8.3.4. CPU_UNCONFIGURE request

This command requests that a guest take offline and unconfigure the specified set of CPUs. The response to this request indicates success or failure for each individually specified CPU.

Before an unconfigure request, a CPU must be part of the logical domain in the hypervisor and must be present in the guest's Machine Description. If either of these conditions are not satisfied, the unconfigure response will indicate that the particular CPU is in the `DR_CPU_STAT_NOT_PRESENT` state. No other assumptions may be made about the state of the CPU before an unconfigure request. In particular, attempts to unconfigure a CPU already in the unconfigured state must succeed.

After a successful unconfigure request, the CPU is in the unconfigured state, which means that it is no longer available for general use by the guest operating system. The CPU is still part of the logical domain in the hypervisor and is still present in the guest's Machine Description. The CPU enters the hypervisor from the guest by means of the `cpu_stop` hypervisor API (Section 13.2.2, “cpu_stop”). Further steps required to reach the unconfigured state is guest operating system specific. See [lddr] for details on the Solaris specific implementation of the unconfigure request.

If the guest provides a service for registering a Machine Description update, that update notification will be provided only after steps have been taken to remove the CPU from the logical domain in the hypervisor and from the guest's Machine Description.

30.8.3.5. CPU_FORCE_UNCONFIG request

This request is equivalent to `CPU_UNCONFIGURE` in that it requests that a guest take offline and unconfigure the specified set of CPUs. In addition however, the guest may choose to implement an override to conditions that may have caused failure for any step of a `CPU_UNCONFIGURE` operation.

Note: For example, whereas Solaris may elect to fail a CPU_UNCONFIGURE for a CPU to which certain processes are bound, it may elect to override and unbind those processes in response to the CPU_FORCE_UNCONFIG request in order to complete the unconfigure or offline operation. Such policy decisions are guest operating system specific.

The response to this request indicates success or failure for each individually specified CPU.

If the guest provides a service for registering a Machine Description update, that update notification will be provided only after steps have been taken to remove the CPU from the logical domain in the hypervisor and from the guest's Machine Description.

30.8.3.6. CPU_STATUS request

This command requests the configuration status of specific CPU(s). The response to this request is guest policy specific and is provided upon this request for informational purposes.

30.8.4. CPU_DR_OK response payload

The CPU_DR_OK response uses the following format. The response header is followed by an array of *num_records* status reports, one for each CPU included in the command request.

Each status report provides information on the result of the requested operation.

The data payload length can be computed from the overall packet length minus the header length and minus the total size of the num_records status report records.

Following the array of status reports is a variable length data section that may be used to hold additional string information specific to a particular CPU. Each status report contains an offset into that data section identifying an additional human-readable NUL-terminated ASCII string when relevant. The offset is specified as the byte offset into the string data section relative to the first byte of the overall CPU DR packet header. The domain services header indicates the overall CPU DR packet length.

The CPU status reports have the following format:

Offset	Size	Field name	Description
0	4	*cpu_id*	Virtual CPU ID
4	4	*result*	Result of the operation
8	4	*status*	Status of the CPU
12	4	*string_off*	String offset relative to the start of the CPU DR response packet

30.8.4.1. CPU_DR_OK Result codes

The *result* field in the per CPU_DR_OK response record details the result of the requested operation on the specified CPU within each status record of the CPU_DR_OK response.

The result codes are defined as follows:

Name	Value	Definition
DR_CPU_RES_OK	0x0	Operation succeeded
DR_CPU_RES_FAILURE	0x1	Operation failed
DR_CPU_RES_BLOCKED	0x2	Operation was blocked

Name	Value	Definition
DR_CPU_RES_CPU_NOT_RESPONDING	0x3	CPU was not responding
DR_CPU_RES_NOT_IN_MD	0x4	CPU not defined in the MD

For DR_CPU_UNCONFIGURE the result code DR_CPU_RES_BLOCKED is equivalent to DR_CPU_RES_FAILURE except that the guest is indicating that the operation may succeed with a subsequent DR_CPU_FORCE_UNCONFIG operation.

30.8.4.2. CPU_DR_OK status codes

The status field in the per CPU_DR_OK response record details the resulting status of the specified CPU after the requested operation.

The status codes are defined as follows:

Name	Value	Definition
DR_CPU_STAT_NOT_PRESENT	0x0	CPU ID does not exist even in the MD
DR_CPU_STAT_UNCONFIGURED	0x1	CPU ID exists in MD, but CPU is not configured for use by the guest
DR_CPU_STAT_CONFIGURED	0x2	CPU is configured for use by the guest

30.8.4.3. CPU DR OK response string

Each response record may optionally include a human readable string so that the guest may return a NUL-terminated ASCII string relevant to each CPU with regard to the requested operation.

If no string is provided the *string_off* field in the response record for a cpu has the value of zero.

30.8.5. CPU DR Error response

The message type DR_CPU_ERROR is returned as a response to a malformed request message.

The DR_CPU_ERROR response has the following format:

Offset	Size	Field name	Description
0	*variable*	*err_msg*	NUL-terminated string

The maximum length of the *err_msg* field including the terminal NUL shall be 1024 characters. The error message will indicate the nature of the failure, such as badly formatted request, intra-guest communication failure, etc.

Note

the new err_msg field of the response message payload applies only to response messages, and then only when the *msg_type* element of the header is DR_CPU_ERROR.

30.9. Memory DR service version 1.0

The ability to add or remove memory from a logical domain is driven from the LDom manager through this domain service.

A unit of memory is referred to as a memory block (mblk) and is described by an address,size pair in byte units.

30.9.1. Service ID

The following service ID should should be added to the Domain Services registry for the memory DR capability.

Service ID	Description
"dr-mem"	Dynamic reconfiguration for memory

Each DR service message consists of a fixed message header and optional packet payload described below. The overall payload length is determined by subtracting the size of the memory DR message header from the entire domain services packet size.

30.9.2. Memory DR message header

All memory DR messages begin with the same header. The message argument in the header and the payload that follows the header depend on the message type.

Offset	Size	Field name	Description
0	4	*msg_type*	Message type
4	4	*msg_arg*	Message argument
8	8	*req_num*	Request number

The overall memory DR protocol consists of a command sent to the client guest that then responds with a reply indicating the overall success of the request.

An error response indicates that the operation was not attempted due to an invalid request. An OK response indicates that the requested operation was attempted. An operation may affect multiple memory blocks (mblk). The result of an attempted operation details the status of each mblk affected by the operation in a separate response record.

The message types identify either a request or a response to a request.

30.9.3. Message types

The following constants are defined for memory DR domain service command identifier values:

Request message types:

Type	Value	ASCII Value	Definition
DR_MEM_CONFIGURE	0x4d43	'MC'	Configure (add) memory
DR_MEM_UNCONFIGURE	0x4d55	'MU'	Unconfigure (remove) memory
DR_MEM_UNCONF_STATUS	0x4d53	'MS'	Get memory unconfigure status
DR_MEM_UNCONF_CANCEL	0x4d4e	'MN'	Cancel memory unconfigure
DR_MEM_QUERY	0x4d51	'MQ'	Query memory info

Response message types:

Type	Value	ASCII Value	Definition
DR_MEM_OK	0x6f	'o'	Request completed successfully

Type	Value	ASCII Value	Definition
DR_MEM_ERROR	0x65	'e'	Request failed (not attempted)

30.9.3.1. Message argument

The *msg_arg* field contents depend on the message payload format and this is described in specific request sections below.

30.9.3.2. Request number

The request number in the message header is a monotonically increasing number that uniquely identifies each request message.

Responses to requests are expected to use the same request number so that they can be paired with their original request.

New requests may be issued without waiting for a response to a preceding request. The underlying transport protocol is responsible to ensure reliable, in-order and unduplicated message packets.

Only one DR_MEM_CONFIGURE or DR_MEM_UNCONFIGURE request can be outstanding, see details below.

Requests are to be processed in the order received.

30.9.3.3. DR_MEM_CONFIGURE request

Request header:

Field name	Value
msg_type	DR_MEM_CONFIGURE
msg_arg	*num_records*
req_num	*req_num*

Request payload record format:

Offset	Size	Field name	Description
0	8	*addr*	Mblk base real address
8	8	*size*	Mblk size in bytes

Note: An implementation may not assume that mblks are arranged in a specific order, and may not assume that mblks are not duplicated, but may assume that the mblks do not intersect.

This command requests that a guest providing this service attempts to add the memory described by a set of mblks that have been dynamically configured into the guest's logical domain. The guest will abort the request upon the first failure to configure a mblk. The response to this request indicates success or failure for each individual mblk specified in the request.

Before a configure request, a mblk must be part of the logical domain in the hypervisor and must be present in the guest's Machine Description. If either of these conditions is not satisfied, the configure response will indicate that the particular mblk is in the DR_MEM_STAT_NOT_PRESENT state.

If the guest has registered a MD update service with the LDom manager, the guest should be notified of a MD update, prior to sending it a memory DR configure request.

After successful completion of a configure operation, a mblk is in the configured state, which means it is available for general use by the guest.

Only one DR_MEM_CONFIGURE request can be outstanding. The guest will return a a response with a status value of DR_MEM_RES_BLOCKED if it receives more than one such request.

30.9.3.4. DR_MEM_UNCONFIGURE request

Request header:

Field name	Value
msg_type	DR_MEM_UNCONFIGURE
msg_arg	*num_records*
req_num	*req_num*

Request payload record format:

Offset	Size	Field name	Description	
0	8	*addr*	Mblk base real address	
8	8	*size*	Mblk size in bytes	

Note: An implementation may not assume that mblks are arranged in a specific order, and may not assume that mblks are not duplicated, but may assume that the mblks do not intersect.

This command requests that a guest providing this service attempt to unconfigure the memory described by a set of mblks in the request. The guest will abort the request upon the first failure to unconfigure a mblk. The response to this request indicates success or failure for each individual mblk specified in the request.

After a successful unconfigure request, a mblk is in the unconfigured state, which means that it is no longer available for general use by the guest operating system. The mblk is still part of the logical domain and is still present in the guest's Machine Description.

If the guest provides a service for registering a Machine Description update, that update notification will be provided only after the unconfigured memory has been removed from the guest's Machine Description.

Only one DR_MEM_UNCONFIGURE request can be outstanding. The guest will return a a response with a status value of DR_MEM_RES_BLOCKED if it receives more than one such request.

DR_MEM_UNCONFIGURE is a long running operation and may generate a lot of I/O activity as modified outgoing pages are flushed to disk. A guest may provide interfaces to track and cancel a DR_MEM_UNCONFIGURE operation, see details below.

30.9.3.5. DR_MEM_UNCONF_STATUS request

Request header:

Field name	Value
msg_type	DR_MEM_UNCONF_STATUS

Field name	Value
msg_arg	0
req_num	*req_num*

This command requests the status of a DR_MEM_UNCONFIGURE command in progress. If there is no outstanding unconfigure operation, a DR_MEM_RES_OK result is returned and *num_records* is set to 0 and no payload follows the response.

30.9.3.6. DR_MEM_UNCONF_CANCEL request

Request header:

Field name	Value
msg_type	DR_MEM_UNCONF_CANCEL
msg_arg	0
req_num	*req_num*

This command requests the guest to cancel an outstanding unconfigure operation. Following successful completion of a cancel operation, all the memory for the mblock currently processed in the outstanding unconfigure request will be reconfigured and available for general use by the guest. Any mblocks already unconfigured will not be affected no further mblocks in the request will be processed. If there is no outstanding unconfigure operation, a DR_MEM_RES_OK result is returned. If an error is encountered during cancel, a DR_MEM_RES_FAILURE result is returned and the cancel request will not affect the outstanding unconfigure operation.

30.9.3.7. DR_MEM_QUERY request

Request header:

Field name	Value
msg_type	DR_MEM_QUERY
msg_arg	*num_records*
req_num	*req_num*

Request payload record format:

Offset	Size	Field name	Units	Description
0	8	*addr*	bytes	Mblk base real address
8	8	*size*	Mblk size	

This command queries the current status of the guest memory layout for the memory blocks specified in the request. The request can be used to determine the memory range within each mblk that may be unconfigured by the guest.

30.9.4. DR_MEM_OK response

The DR_MEM_OK response uses the following format. The response header may be followed by an array of *num_records* status reports, one for each record included in the command request. Each status report provides information on the results of the requested operation.

The data payload length can be computed from the overall packet length minus the header length and minus the total size of the *num_records* status report records.

Following the array of status reports for certain requests is a variable length data section that may be used to hold additional string information specific to a particular mblk. Each status report contains an offset into that data section identifying an additional human readable NUL-terminated ASCII string when relevant. The offset is specified as the byte offset relative to the first byte of the overall MEM DR packet header. The domain services header indicates the overall memory DR packet length.

30.9.4.1. DR_MEM_OK result codes

The result code in a DR_MEM_OK response details the affect of the attempted operation.

The result codes are defined as follows:

Name	Value	Definition
DR_MEM_RES_OK	0x0	Operation succeeded
DR_MEM_RES_FAILURE	0x1	Operation failed
DR_MEM_RES_BLOCKED	0x2	Operation was blocked
DR_MEM_RES_CANCELED	0x3	CPU was not responding
DR_MEM_RES_NOWORK	0x4	CPU not defined in the MD
DR_MEM_RES_PERM	0x5	Permanent memory in span

30.9.4.2. DR_MEM_OK status code

The *status* field in a DR_MEM_OK response record details the resulting status of the specified mblk after the requested operation.

The status codes are defined as follows:

Name	Value	Definition
DR_MEM_STAT_NOT_PRESENT	0x0	Mblk does not exist in MD
DR_MEM_STAT_UNCONFIGURED	0x1	Mblk exists in MD, but mblk is not configured for use by guest
DR_MEM_STAT_CONFIGURED	0x2	Operation was blocked

30.9.4.3. DR_MEM_OK response string

Each response record may optionally include a human readable string so that the guest may return a NUL-terminated ASCII string relevant to each mblk with regard to the requested operation. If no string is provided the *string_off* field in the response record for a mblk has the value of zero.

30.9.4.4. DR_MEM_CONFIGURE response payload

Response header:

Field name	Value
msg_type	DR_MEM_OK
msg_arg	*num_records*

Field name	Value
req_num	*req_num*

Response payload record format:

Offset	Size	Field name	Description
0	8	*addr*	Mblk address
8	8	*size*	Mblk size in bytes
16	4	*result*	Result of the operation
20	4	*status*	Status of the mblk
24	4	*string_off*	String offset relative to start

Result codes:

```
DR_MEM_RES_OK
DR_MEM_RES_FAILURE
DR_MEM_RES_BLOCKED
DR_MEM_RES_NOWORK
```

Status codes:

```
DR_MEM_STAT_NOT_PRESENT
DR_MEM_STAT_CONFIGURED
DR_MEM_STAT_UNCONFIGURED
```

30.9.4.5. DR_MEM_UNCONFIGURE response payload

Response header:

Field name	Value
msg_type	DR_MEM_OK
msg_arg	*num_records*
req_num	*req_num*

Response payload record format:

Offset	Size	Field name	Description
0	8	*addr*	Mblk address
8	8	*size*	Mblk size in bytes
16	4	*result*	Result of the operation
20	4	*status*	Status of the mblk
24	4	*string_off*	String offset relative to start

Result codes:

```
DR_MEM_RES_OK
DR_MEM_RES_FAILURE
DR_MEM_RES_BLOCKED
DR_MEM_RES_CANCELED
```

DR_MEM_RES_NOWORK
DR_MEM_RES_PERM

Status codes:

DR_MEM_STAT_CONFIGURED
DR_MEM_STAT_UNCONFIGURED

30.9.4.6. DR_MEM_UNCONF_STATUS response payload

Response header:

Field name	Value
msg_type	DR_MEM_OK
msg_arg	*num_records*
req_num	*req_num*

Response payload record format:

Offset	Size	Field name	Description
0	8	*total*	Total region size
8	8	*collected*	Amount of collected memory

Result codes:

DR_MEM_RES_OK

30.9.4.7. DR_MEM_UNCONF_CANCEL response payload

Response header:

Field name	Value
msg_type	DR_MEM_OK
msg_arg	*result*
req_num	*req_num*

Result codes:

DR_MEM_RES_OK
DR_MEM_RES_FAILURE

30.9.4.8. DR_MEM_QUERY response payload

Response header:

Field name	Value
msg_type	DR_MEM_OK
msg_arg	*num_records*
req_num	*req_num*

Response payload record format:

Offset	Size	Field name	Description
0	8	*addr*	Mblk address
8	8	*size*	Mblk size in bytes
16	8	*perm*	Amount of permanent memory in mblk
24	8	*first_perm*	First permanent RA in mblk
32	8	*last_perm*	Last permanent RA in blk

The *addr* and *size* fields equals the values in the corresponding request.

The *perm* field is the size of the permanent memory within the mblk. Memory which may not be unconfigured is referred to as permanent memory.

The *first_perm* field is the lower bound of the permanent memory range within the mblk.

The *last_perm* field is the upper bound of the permanent memory range within the mblk.

The memory in the mblk not contained within the lower and upper bounds is removable. If the *perm* size in the response is less than difference of *first_perm* and *last_perm* addresses, it is indicative that there is additional removable (non-permanent) memory between the lower and upper bounds.

Result codes:

DR_MEM_RES_OK

30.9.5. DR_MEM_ERROR response

The message type DR_MEM_ERROR is returned as a response to a malformed request message.

Response header:

Field name	Value
msg_type	DR_MEM_ERROR
msg_arg	0
req_num	*req_num*

Response payload record format:

Offset	Size	Field name	Description
0	*variable*	*err_msg*	NUL-terminated string

The maximum length of the *err_msg* field including the terminal NUL shall be 1024 characters. The error message will indicate the nature of the failure, such as badly formatted request, intra-guest communication failure, etc.

Note

the new err_msg field of the response message payload applies only to response messages, and then only when the *msg_type* element of the header is DR_MEM_ERROR.

30.10. VIO DR service version 1.0

This service provides ability for the logical domain manager to request the addition or removal of virtual devices. The service is called Virtual I/O Dynamic Reconfiguration (VIO DR).

This mechanism, if supported by the guest operating system in a virtual machine, allows the logical domain manager to remotely reconfigure the virtual IO resources provided by and used by a guest domain without that guest domain needing to be rebooted to “discover” those resources.

30.10.1. Service ID

The following service ID if exported by a guest domain indicates that the guest supports VIO DR and the domain service described in this section.

Service ID	Description
`"dr-vio"`	Dynamic reconfiguration for virtual I/O devices

30.10.2. Message format

Payload:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	8	*dev_id*	Device ID
16	4	*msg_type*	Message type
20	*variable*	*name*	Device name

30.10.3. Message types

The message type (*msg_type*) field contains a value indicating the type of operation being requested. The following constants are defined for VIO DR Domain Service as message type values:

Type	Value	ASCII Value	Definition
DR_VIO_CONFIGURE	0x494f43	'IOC'	Configure a new device
DR_VIO_UNCONFIGURE	0x494f55	'IOU'	Unconfigure a device
DR_VIO_FORCE_UNCONFIG	0x494f46	'IOF'	Forcibly unconfigure a device
DR_VIO_STATUS	0x494f53	'IOS'	Request the status of a device

30.10.3.1. DR_VIO_CONFIGURE request

This command requests that a guest providing this service attempt to configure and bring online a virtual I/O device that has been dynamically added or configured into the logical domain. The response to this request indicates success or failure for this attempt.

Before a configure request, the selected device must be part of the logical domain's machine description. No other assumptions may be made about the state of the device before a configure request. In particular, attempts to configure a device already in the configured state must succeed. This service supports adding

new virtual IO devices under the `channel-devices` node of the MD, but not directly under its parent, the `virtual-devices` node.

If the guest registers a service for notifying it of a Machine Description update, that update notification must be provided to the guest prior to the configure request being given.

After a successful configure request, the device is in the configured state, which means that it is available for general use by the guest. Further steps required to reach the configured state is guest operating system specific.

30.10.3.2. `DR_VIO_UNCONFIGURE` request

This command requests that a guest take offline and unconfigure the specified device. The response to this request indicates success or failure of the request.

Before an unconfigure request, a device must be part of the logical domain's machine description. No other assumptions may be made about the state of the device before an unconfigure request. In particular, attempts to unconfigure a device already in the unconfigured state must succeed.

After a successful unconfigure request, the device is in the unconfigured state, which means that it is no longer available for general use by the guest operating system. The device is still present in the guest's Machine Description. The steps required to reach the unconfigured state is guest operating system specific.

If the guest provides a service for registering a Machine Description update that update notification will be provided only after steps have been taken to remove the device from the logical domain in the hypervisor and from the guest's Machine Description.

30.10.3.3. `DR_VIO_FORCE_UNCONFIG` request

This request is equivalent to `DR_VIO_UNCONFIGURE` in that it requests that a guest take offline and unconfigure the specified device. In addition however, the guest may choose to implement an override to conditions that may have caused failure for any step of a `DR_VIO_UNCONFIGURE` operation.

The response to this request indicates success or failure of the request. If the guest provides a service for registering a Machine Description update, that update notification will be provided only after steps have been taken to remove the device from the logical domain in the hypervisor and from the guest's Machine Description.

30.10.3.4. `DR_VIO_STATUS` request

This command requests the configuration status of a specific device. The response to this request indicates the current state of the device, which can include an optional descriptive string.

30.10.3.5. Request number

The request number in the message header is a monotonically increasing number that uniquely identifies each request message.

Responses to requests are expected to use the same request number so they can be paired with their original request.

New requests may be issued without waiting for a response to a preceding request. The underlying transport protocol is responsible to ensure reliable, in-order and unduplicated message packets.

Requests are to be processed in the order received.

30.10.3.6. Device Name

This element of the request message identifies the type of the device which is the target of the request. The Device Name *name* field in the request message corresponds to the `name` property of the `virtual-device` node in the Machine Description. It consists of a `NUL`-terminated string. The maximum length of this string is 256 characters, including the terminating `NUL`.

30.10.3.7. Device ID

The Device ID *dev_id* field in the request message corresponds to the `cfg-handle` of the `virtual-device` node in the guest's Machine Description.

30.10.4. VIO DR response message

The overall VIO DR protocol consists of a command sent to the client guest which then responds with a reply indicating the result of the request.

30.10.4.1. VIO DR response message format

The VIO DR response message has the following format:

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*result*	Result code
12	4	*status*	Status code
16	*variable*	*reason*	Reason string (cause of error)

30.10.4.2. VIO DR Result codes

The result field in the above message indicates the result of the requested operation on the specified device. The result codes are defined as follows:

The status codes are defined as follows:

Name	Value	Definition
DR_VIO_RES_OK	0x0	Operation succeeded
DR_VIO_RES_FAILURE	0x1	Operation failed
DR_VIO_RES_BLOCKED	0x2	Operation was blocked
DR_VIO_RES_NOT_IN_MD	0x3	Device undefined in the MD

For a `DR_VIO_UNCONFIGURE` request the result code `DR_VIO_RES_BLOCKED` is equivalent to `DR_VIO_RES_FAILURE` except that the guest is indicating that the operation may succeed with a subsequent `DR_VIO_FORCE_UNCONFIG` operation.

30.10.4.3. VIO DR status codes

The *status* field in the response message indicates the resulting status of the specified device after the requested operation. For the response message to a configure or unconfigure request, the *result* field indicates the outcome of the operation. The *status* field contains one of the status codes below to indicate state of the device after the attempted operation.

For the response message corresponding to a successful DR_VIO_STATUS request, the *status* field will contain one of the codes below, and the *result* field will contain DR_VIO_RES_OK. If the DR_VIO_STATUS operation fails, the *result* field will contain DR_VIO_RES_FAILURE and the *status* field will not be meaningful.

The status codes are defined as follows:

Name	Value	Definition
DR_VIO_STAT_NOT_PRESENT	0x0	Device does not exist in the MD
DR_VIO_STAT_UNCONFIGURED	0x1	Device exists in the MD, but is not configured for use by the guest
DR_VIO_STAT_CONFIGURED	0x2	Device is configured for use by the guest

A VIO device in the DR_VIO_STAT_UNCONFIGURED state may be safely removed from the domain configuration. Conversely, a VIO device in the DR_VIO_STAT_CONFIGURED state must not be removed from the domain configuration as the guest may be accessing it.

30.10.4.4. VIO DR “reason” string

The response message may optionally include a human-readable string so that the guest may return a NUL-terminated ASCII string containing additional information regarding the requested operation. The maximum length of this string is 1024 characters including the terminating NUL.

If there is no “reason” string, this field shall contain a single NUL character at the start of the field. In the case of a successful operation no response string will be returned.

30.11. Crypto DR service version 1.0

The ability to dynamically add or remove hardware crypto providers from a logical domain is driven from the LDom manager through this domain service. Separate services will be defined for the Modular Arithmetic Unit (MAU) and the Control Word Queue (CWQ) hardware components.

30.11.1. Service ID

The following service IDs correspond to the cryptographic unit dynamic reconfiguration capabilities of a guest operating system.

Service ID	Description
"dr-crypto-mau"	Dynamic reconfiguration for MAU devices
"dr-crypto-cwq"	Dynamic reconfiguration for CWQ devices

30.11.2. Message format header

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*msg_type*	Message type

The same DR service messages are used for both services. Each message consists of a fixed message header and payload as described below. Overall, the Crypto DR service messages are very similar to the CPU DR messages.

All Crypto DR messages begin with the same header. The payload that follows the header is specific to a particular message type. The Crypto DR protocol consists of a command sent to the client guest which then responds with a reply indicating the success or failure of the request.

30.11.3. Message Types

The following message types are defined for the Crypto DR domain service.

30.11.3.1. Request messages

Type	Value	ASCII Value	Definition
DR_CRYPTO_CONFIGURE	0x43	'C'	Configure a new crypto unit
DR_CRYPTO_UNCONFIGURE	0x55	'U'	Unconfigure a crypto unit
DR_CRYPTO_FORCE_UNCONFIG	0x46	'F'	Forcibly unconfigure a crypto unit
DR_CRYPTO_STATUS	0x53	'S'	Request the status of a crypto unit

30.11.3.2. Response messages

Type	Value	ASCII Value	Definition
DR_CRYPTO_OK	0x6f	'o'	Request completed successfully
DR_CRYPTO_ERROR	0x65	'e'	Request failed

30.11.4. Request Payload

The Crypto DR requests all use the same payload format, which is a list of records of virtual CPU IDs within a guest. Because there is no crypto unit ID defined in the guest, a virtual CPU ID which maps to the desired crypto unit is passed as the identifier. There should be one virtual CPU ID specified per targeted crypto unit.

The payload is as follows:

Offset	Size	Field name	Description
0	4	*id0*	First virtual CPU ID
4	4	*id1*	Second virtual CPU ID
8	4	*id2*	Third virtual CPU ID
...	4	*idN*	*N*th virtual CPU ID

30.11.5. Request Number

The request number is a monotonically increasing value that uniquely identifies each request. Responses to requests are expected to use the same request number so they can be paired with the original request. Requests are to be processed in the order received.

30.11.6. DR_CRYPTO_CONFIG request

This command requests that a guest attempt to configure and bring online the crypto units associated with the set of virtual CPU ID supplied in the request message. In order to be successful, the crypto unit

and associated virtual CPUs must already exist in the guest's Machine Description (MD). If both of these conditions are not satisfied, an error is returned.

30.11.7. `DR_CRYPTO_UNCONFIG` request

This command requests that the guest attempt to offline and unconfigure the targeted crypto units. An associated virtual CPU ID is supplied in the request message to identify the crypto unit. In order to be successful, the crypto unit and associated virtual CPUs must already exist in the guest's Machine Description (MD). If both of these conditions are not satisfied, an error is returned.

30.11.8. `DR_CRYPTO_FORCE_UNCONFIG` request

This command requests that the guest forcibly attempt to offline and unconfigure the targeted crypto units. However, there is no still guarantee that the guest will be able to successfully complete the request.

30.11.9. `DR_CRYPTO_STATUS`

The command requests the configuration status for specific crypto units.

30.11.10. `DR_CRYPTO_OK` response payload

The `DR_CRYPTO_OK` response uses the following format. The response header is followed by an array of status reports, one for each crypto unit targeted in the command request. Each status report provides information on the result of the requested operation. Because there is no crypto unit ID, the virtual CPU ID is carried in the status report. The crypto unit status reports have the following format:

Offset	Size	Field name	Description
0	4	*cpuid*	Virtual CPU ID
4	4	*result*	Result of the operation
8	4	*status*	Status of the crypto unit

30.11.11. `DR_CRYPTO_OK` result codes

The *result* field in the per crypto unit response record conveys the result of the requested operation for that crypto unit. The result codes are defined as follows:

Name	Value	Definition
DR_CRYPTO_RES_OK	0x0	Operation succeeded
DR_CRYPTO_RES_FAILURE	0x1	Operation failed
DR_CRYPTO_RES_BAD_CPU	0x2	CPU not in the MD
DR_CRYPTO_RES_BAD_CRYPTO	0x3	Crypto unit not in the MD

30.11.12. DR CRYPTO OK status codes

The *status* field in the per crypto unit response record conveys the configuration status for the targeted crypto unit. The status codes are defined as follows:

Name	Value	Definition
DR_CRYPTO_STAT_NOT_PRESENT	0x0	Crypto unit does not exist in the MD
DR_CRYPTO_STAT_UNCONFIGURED	0x1	Crypto unit exists in the MD, but is not configured for use by the guest
DR_CRYPTO_STAT_CONFIGURED	0x2	Crypto unit is configured for use by the guest

30.11.13. DR Crypto Error Response

The message type `DR_CRYPTO_ERROR` is returned as the response to a malformed request message. No additional payload is provided.

30.11.14. Operational Overview

30.11.14.1. Offlining a Crypto Unit

When the LDom manager decides to offline a crypto unit (or multiple crypto units), it will build `DR_CRYPTO_UNCONFIG` domain service messages, including a list of virtual CPU IDs, each associated with the specific crypto unit being taken offline. This message must be sent and acknowledged in advance of any change to the machine description.

The domain service peers in the guest must guarantee that all jobs have completed for that crypto unit and that no additional work will be scheduled before responding successfully.

30.11.14.2. Onlining a Crypto Unit

When the LDom manager decides to online a crypto unit, if it is a new crypto unit, the guest must first gets an MD update which includes information about the new crypto unit.

Once that has occurred, the LDom manager will build `DR_CRYPTO_CONFIG` domain service messages, including a list of virtual CPU IDs, each associated with the specific crypto unit being brought online.

The domain service peers in the guest will re-read the MD and configure in the new crypto unit based on the virtual CPU IDs included in the `DR_CRYPTO_CONFIG` message payload. Once the configuration has completed, the response will be returned to the LDom manager.

30.12. Variable Configuration version 1.0

The Variable Configuration capability provides the ability for a guest to update the LDom variable store that is managed by the LDom manager or SP.

30.12.1. Service IDs

There are two service IDs defined to support LDom variable updates, one that describes a primary service and one that describes a backup service. In the event that the primary service is not available, the guest can fall back to using the backup service. The backup service uses the identical protocol as the primary service but is subordinate in priority to the primary service.

Implementation Note: The LDom manager provides the primary service. In the case where the LDom manager has not been started, or is not currently running, variable updates can be communicated to the SP using the backup service. OpenBoot in the control domain will use the backup service since the LDom manager will not be running. OpenBoot in all other domains will use the primary service as long as the LDom manager is available.

The following service IDs should should be added to the Domain Services registry for the LDom variables capability.

Service ID	Description
`"var-config"`	Primary LDoms variable management
`"var-config-backup"`	Secondary LDoms variable management

30.12.2. Message Header

Offset	Size	Field name	Description
0	4	*cmd*	Command

30.12.3. Message types

The following constants are defined for Variable Configuration domain service command identifier values:

Name	Value	Definition
VAR_CONFIG_SET_REQ	0x0	Request setting a variable
VAR_CONFIG_DELETE_REQ	0x1	Request deleting a variable
VAR_CONFIG_SET_RESP	0x2	Response to a set request
VAR_CONFIG_DELETE_RESP	0x3	Response to a delete request

30.12.4. Set Variable Payload

The *set* command updates the variable in the store. If the variable already exists in the store, the new value replaces the old value. If the variable does not exist in the store, it is added.

The Variable Configuration header is followed by two NUL-terminated strings. The first represents the name of the variable to set. The second represents the value to set it to.

Offset	Size	Field name	Description
0	*variable*	*name*	Name of variable to set
variable	*variable*	*value*	Value of variable

30.12.5. Delete Variable Payload

The *delete* command removes a variable from the store. The Variable Configuration header is followed by one NUL-terminated string. The string represents the name of the variable to delete.

Offset	Size	Field name	Description
0	*variable*	*name*	Name of variable to delete

30.12.6. Response Payload

Responses to set and delete commands share the same format. The Variable Configuration header is followed by the following response payload:

Offset	Size	Field name	Description
0	4	*result*	Result of operation

30.12.6.1. Response Result Codes

The result field in the response payload details the result of the requested operation. The result codes are defined as follows:

Name	Value	Definition
VAR_CONFIG_SUCCESS	0x0	Operation succeeded

Name	Value	Definition
VAR_CONFIG_NO_SPACE	0x1	Variable store is full
VAR_CONFIG_INVALID_VAR	0x2	Invalid variable format
VAR_CONFIG_INVALID_VAL	0x3	Invalid value format
VAR_CONFIG_VAR_NOT_PRESENT	0x4	Variable not present to delete

30.13. Security key domain service version 1.0

The Security Key storage domain service provides the ability for a guest to update the Security Key storage that is managed by the LDom manager or system controller (SC) (aka service processor).

30.13.1. Service IDs

There are two service IDs defined to support Security key storage, one that describes a primary service and one that describes a backup service. In the event that the primary service is not available, the control domain can fall back to using the backup service. The backup service uses the identical protocol as the primary service but is subordinate in priority to the primary service.

Service ID	Description
"keystore"	Primary security key management
"keystore-backup"	Secondary security key management

Programming Note

The LDom manager typically provides the primary service and the SC can provide the backup service. For example, OpenBoot in the Control Domain can use the backup service to the SC as the LDom manager will typically not be running when OpenBoot is active. All other domains will use the primary service as long as the LDom manager is available.

30.13.2. Message Header

Offset	Size	Field name	Description
0	4	*cmd*	Command

30.13.3. Message types

The following constants are defined for Security Key Store domain service command identifier values:

Name	Value	Definition
KEYSTORE_SET_REQ	0x0	Request setting a security key
KEYSTORE_DELETE_REQ	0x1	Request deleting a security key
KEYSTORE_SET_RESP	0x2	Response to a set request
KEYSTORE_DELETE_RESP	0x3	Response to a delete request

30.13.3.1. Set keystore Payload

The *set* command updates the security key in the store. If the security key already exists in the store, the new value replaces the old value. If the security key does not exist in the store, it is added. The Security Key header is followed by two NUL-terminated strings. The first represents the name of the Security Key to set. The second represents the value to set it to.

Offset	Size	Field name	Description
0	*variable*	*name*	Name of security key to set
variable	*variable*	*value*	Value of security key

30.13.3.2. Delete keystore Payload

The *delete* command removes a Security Key from the store. The Security Key header is followed by one NUL-terminated string. The string represents the name of the Security Key to delete.

Offset	Size	Field name	Description
0	*variable*	*name*	Name of security key to delete

30.13.4. Response Payload

Responses to set and delete commands share the same format. The Security Key header is followed by the following response payload:

Offset	Size	Field name	Description
0	4	*result*	Result of operation

30.13.4.1. Response Result Codes

The *result* field in the response payload details the result of the requested operation. The result codes are defined as follows:

Name	Value	Definition
KEYSTORE_SUCCESS	0x0	Operation succeeded
KEYSTORE_NO_SPACE	0x1	Security key store is full
KEYSTORE_INVALID_NAME	0x2	Invalid security key name format
KEYSTORE_INVALID_VAL	0x3	Invalid security key value format
KEYSTORE_NOT_PRESENT	0x4	Security key not present to delete

30.14. PRI Domain Service 1.0

The PRI is intended to contain various kinds of information about a system. Much of this information has previously been contained in the Machine Description (MD). The PRI domain service will facilitate access to this information from the Control Domain.

The SC will generate the PRI and send PRI Update Messages containing the PRI to the Control domain using domain service. The messages will generally be sent when the initial domain service registration occurs. They may also be sent asynchronously on a channel when the PRI is updated by the SC. Control domain will respond to the PRI update message by sending a PRI update response. See following sections for the details.

30.14.1. Service ID

Service ID	Description
"pri-update"	PRI update domain service

30.14.2. PRI Update Message

Offset	Size	Field name	Description
0	8	*pri_msgnum*	Message number
8	8	*pri_size*	Size of the PRI data
16	*variable*	*pri*	PRI data

30.14.3. PRI Update Response

Offset	Size	Field name	Description
0	8	*pri_msgnum*	Message number responding to
8	8	*status*	Status of PRI update

30.14.4. pri_msgnum

The *pri_msgnum* is the message number in the PRI Update header which is a monotonically increasing number that uniquely identifies each message. Responses to messages are expected to use the same message number so that they can be paired with their original message.

New messages may be issued without waiting for a response to a preceding message. The underlying transport protocol is responsible to ensure reliable, in-order and unduplicated message packets.

Messages are to be processed in the order received.

30.14.5. Response Status Codes

Name	Value	Definition
PRI_UPDATE_ACK	0x0	PRI successfully received and verified
PRI_UPDATE_INVALID_MSG	0x1	Message payload invalid
PRI_UPDATE_INVALID_PRI	0x2	PRI format invalid

30.15. System Info version 1.0

The System Information capability provides for the ability to query various system information, such as POSIX utsname strings. It is anticipated that this service may be expanded as needed in the future to support such areas as, for example, Solaris sysinfo and Solaris kstats.

30.15.1. Service ID

Service ID	Description
"system-info"	Guest system information queries

30.15.2. Message header

Offset	Size	Field name	Description
0	4	*cmd*	Command

30.15.3. Message types

The following constants are defined for System Information domain service command identifier values.

Name	Value	Definition
SYS_INFO_GET_REQ	0x0	Request to get system information
SYS_INFO_GET_RESP	0x1	Response to a “get” request

30.15.4. Get Information Payload

The Get Information header is followed by a variable number of NUL-terminated strings. Each string represents a specific piece of system information to return. The following are the supported strings:

Name	Description
"uts-sysname"	utsname.sysname
"uts-nodename"	utsname.nodename
"uts-release"	utsname.release
"uts-version"	utsname.version
"uts-machine"	utsname.machine

Note that the string values returned should be identical to what is provided by the POSIX uname(1) user command. Non-POSIX compliant guest systems may choose to support any of these items if they are appropriate.

30.15.5. Get Information Response Payload

Response to the Get Information request is a message with the Get Information header followed by the following response payload. In the event that no datum names were identified, the result field will be SYS_INFO_INVALID_NAME and the num_pairs field will be zero. If multiple datum names were provided and not all of them were matched, the result field will be SYS_INFO_SUCCESS and the number of datum names matched will be in the *num_pairs* field and their corresponding name/value pairs will be in the *name*/*value* fields. Each datum name and datum value will be a NUL-terminated string.

Offset	Size	Field name	Description
0	4	*result*	Result of Operation
4	4	*num_pairs*	Number of name/value pairs returned
8	*variable*	*name1*	Name of first datum
variable	*variable*	*value1*	value of first datum
variable	*variable*	*namen*	Name of n'th datum
variable	*variable*	*valuen*	value of n'th datum

30.15.6. Response Result Codes

The result field in the response payload details the results of the requested operation. The result codes are defined as follows:

Name	Value	Definition
SYS_INFO_SUCCESS	0x0	Operation succeeded
SYS_INFO_INVALID_NAME	0x1	Invalid datum name supplied

30.16. SNMP service version 1.0

This case describes the domain service interface through which a client communicates with the system controller (SC) (aka service processor) using the SNMP protocol. This service can be used to access environmental data and other information that may be exported from the system controller. This information can be dynamically updated— it is the responsibility of a guest operating system to monitor and provide access to this information to it's users if so desired. Such presentation interfaces are beyond the scope of this document.

Each of the SNMP Messages consists of a header and a payload. The headers are defined by this specification and the payloads consist of data encoded according to the SNMP protocol, as defined by a number of IETF RFCs. The SNMP protocol versions supported and their message formats are not part of this specification. The version support is negotiated between the guest operating system's driver and the SNMP Agent resident on the SC. The SNMP PDUs are simply encapsulated by the SNMP Domain Service that is the subject of this specification. The length of the SNMP PDU is encoded in the message itself and is not part of the header. It is left to the consumers at the endpoints on this domain service to send and receive and detect properly formed SNMP messages. The domain service described below is version 1.0.

30.16.1. Service ID

Service ID	Description
`"snmp"`	SNMP domain service

30.16.2. Message header

Offset	Size	Field name	Description
`0`	`8`	*`number`*	Message number
`8`	`8`	*`type`*	Message type
`16`	*variable*	*`payload`*	Message-dependent payload

All SNMP messages have the same header format consisting of a message *`number`* and a message *`type`*.

The message *`number`* is a monotonically increasing number that uniquely identifies each message. Responses to messages are expected to use the same message number so that they can be paired with their original message. The message number may also be used to distinguish between multiple instances of the same message type.

New messages may be issued without waiting for a response to a preceding message. The underlying transport protocol is responsible for ensuring reliable, in-order and unduplicated message packets.

Messages are to be processed in the order received.

30.16.3. Message types

The message *`type`* is used to distinguish the different message types. There are three types defined in this initial version of the protocol specification.

Name	Value	Definition
`SNMP_REQUEST`	`0x0`	SNMP request to an SNMP agent
`SNMP_REPLY`	`0x1`	Message from an SNMP agent
`SNMP_ERROR`	`0x2`	Error message fro an SNMP agent

30.16.3.1. SNMP Request Message

An SNMP_REQUEST message is sent by the client carrying a payload to be delivered to the SNMP agent.

The message *number* value will be used in the SNMP_REPLY message sent in response to this request.

The message *type* field should indicate an SNMP_REQUEST.

The *payload* field has a variable length depending on the SNMP data sent as part of the request.

30.16.3.2. SNMP Reply Message

An SNMP_REPLY message is sent by the server in response to a request from the client. It carries a payload whose content is determined by the SNMP agent acting on the request.

The *number* field contains the value in the *number* field of the original request being serviced.

The message *type* field should indicate an SNMP_REPLY.

The *payload* field has a variable length depending on the SNMP data.

30.16.3.3. SNMP Error Message

An SNMP_ERROR message is sent by the server in response to a request from the client that cannot be serviced. These include errors such as being unable to contact the SNMP Agent or timing out waiting for a reply from the SNMP agent.

The SNMP_ERROR message has no payload.

The message *number* field contains the value in the message *number* field of the request that could not be serviced.

The message *type* field should indicate an SNMP_ERROR.

30.17. Domain Suspend service version 1.0

The Domain Suspend domain service allows Solaris to initiate a domain suspend operation at the request of the domain manager. This is required for Cooperative Guest migration wherein the guest initiates the suspend operation by calling into the hypervisor. Existing Warm Migration makes use of CPU DR to reduce a domain to 1-strand before the hypervisor pauses the domain. In order to eliminate CPU DR from migration and to eliminate other restrictions on Warm Migration, Solaris on the guest will call into the hypervisor to initiate suspension of the domain. Hypervisor calls to support the guest initiated suspend are described in: Section 12.1.5, “mach_suspend”, Section 13.2.11, “cpu_tick_npt”, and Section 13.2.12, “cpu_stick_npt”.

30.17.1. Service ID

Service ID	Description
"domain-suspend"	Sequence a suspend operation on the domain

30.17.2. Domain Suspend Request

Offset	Size	Field name	Description
0	8	*req_num*	Request number

Offset	Size	Field name	Description
8	8	*type*	Message type

Only one message type is supported and therefore there is only one valid value for the *type* field. However, a *type* field is used to allow for more request types to be added in the future without changing the request format.

30.17.3. Message types

Name	Value	Definition
DOMAIN_SUSPEND_SUSPEND	0x0	Request to suspend

30.17.3.1. Domain Suspend Reply Message

Offset	Size	Field name	Description
0	8	*req_num*	Request number
8	4	*result*	Result of operation
12	4	*rec_result*	Result of recovery operation
16	*variable*	*reason_result*	NUL-terminated ASCII string describing reason for error

The NUL-terminated reason string is limited to 512 characters in length, including the NUL terminator.

When the reason field begins with its first byte being the NUL terminator, it is said to be set to the empty string.

30.17.3.2. Domain Suspend response result values

The following constants are defined for the suspend service response *result* and *rec_result* values.

Result values:

Name	Value	Definition
DOMAIN_SUSPEND_PRE_SUCCESS	0x0	Pre-suspend success
DOMAIN_SUSPEND_PRE_FAILURE	0x1	Pre-suspend failure
DOMAIN_SUSPEND_INVALID_MSG	0x2	The request is invalid
DOMAIN_SUSPEND_INPROGRESS	0x3	Existing suspend is in progress
DOMAIN_SUSPEND_FAILURE	0x4	Suspend failure
DOMAIN_SUSPEND_POST_SUCCESS	0x5	Post-suspend success
DOMAIN_SUSPEND_POST_FAILURE	0x6	Post-suspend failure

Recovery Result values:

Name	Value	Definition
DOMAIN_SUSPEND_REC_SUCCESS	0x0	Recovery success
DOMAIN_SUSPEND_REC_FAILURE	0x1	Recovery failure

For responses with a result value of DOMAIN_SUSPEND_PRE_FAILURE, DOMAIN_SUSPEND_FAILURE, or DOMAIN_SUSPEND_POST_FAILURE, the *reason* field

may be populated with a NUL-terminated string describing the reason for the failure. This is optional and when a NUL-terminated reason string is not provided, the *reason* field must must be set to the empty string. For all other response messages (DOMAIN_SUSPEND_PRE_SUCCESS, DOMAIN_SUSPEND_INVALID_MSG, DOMAIN_SUSPEND_INPROGRESS, and DOMAIN_SUSPEND_POST_SUCCESS) the *reason* field is not used and should be set to the empty string.

For responses with a result value of DOMAIN_SUSPEND_PRE_FAILURE or DOMAIN_SUSPEND_FAILURE the *rec_result* must be set to either DOMAIN_SUSPEND_REC_SUCCESS or DOMAIN_SUSPEND_REC_FAILURE. For all other response messages (DOMAIN_SUSPEND_PRE_SUCCESS, DOMAIN_SUSPEND_INVALID_MSG, DOMAIN_SUSPEND_INPROGRESS, and DOMAIN_SUSPEND_POST_SUCCESS, and DOMAIN_SUSPEND_POST_FAILURE) the *rec_result* field is not used and its value should be DOMAIN_SUSPEND_REC_SUCCESS.

30.17.4. Domain Suspend request handling

30.17.4.1. Invalid Request

Upon receiving a suspend domain request, the guest domain will confirm that the request is a DOMAIN_SUSPEND_SUSPEND request type. If the request has a *type* field other than DOMAIN_SUSPEND_SUSPEND, the guest will send a response message with result value DOMAIN_SUSPEND_INVALID_MSG and a *req_num* equal to request *req_num* and then take no further action in response to the message.

30.17.4.2. Suspend in Progress

If the request is a valid DOMAIN_SUSPEND_SUSPEND request, but the guest is already processing a suspend request, the guest will send a response message with *result* value DOMAIN_SUSPEND_INPROGRESS and *req_num* value equal to the *req_num* received in the newer request. The guest will take no further action in response to the message.

30.17.4.3. Pre-suspend

The guest will then perform any pre-suspend processing which results in either success or failure. In the event of success, the guest will send a response message with *result* value DOMAIN_SUSPEND_PRE_SUCCESS and a *req_num* value equal to the request *req_num*. In the event of failure, the guest will undo any partial pre-suspend processing that successfully completed and then send a response message with result value DOMAIN_SUSPEND_PRE_FAILURE, a *req_num* value equal to the request *req_num*, and optionally a NUL-terminated *reason* string describing the reason for the failure. If the partial pre-suspend processing is successfully undone, the *rec_result* field will be set to DOMAIN_SUSPEND_REC_SUCCESS. Otherwise, the *rec_result* field will be set to DOMAIN_SUSPEND_REC_FAILURE. The guest will then take no further action in response to the request. The intent here is for the user to be presented with a warning message derived from the reason field indicating that the pre-suspend processing failed and (if applicable) that a particular recovery operation failed. If a recovery operation failed, the user must then inspect the guest domain and take any action required to cleanup after failed recovery.

30.17.4.4. Suspend

Next, after the guest sends the DOMAIN_SUSPEND_PRE_SUCCESS response, it will attempt to suspend itself using the hypervisor interface described in Section 12.1.5, “mach_suspend” which results in either success or failure. In the event of success, the guest will be suspended and will do no further processing until it is resumed.

Implementation Note

Although not described here, an out-of-band mechanism exists allowing the domain manager to query the state of the domain and to determine when the guest has successfully suspended itself. It is expected that the domain manager will monitor the guest state until the guest state indicates that the guest is suspended OR until a DOMAIN_SUSPEND_FAILURE response is received.

In the event that the guest fails to suspend, the guest will undo the pre-suspend processing and then send a response message with *result* value DOMAIN_SUSPEND_FAILURE, a *req_num* value equal to the request *req_num*, and optionally a NUL-terminated *reason* string describing the reason for the failure. If the pre-suspend processing is successfully undone, the *rec_result* field will be set to DOMAIN_SUSPEND_REC_SUCCESS. Otherwise, the *rec_result* field will be set to DOMAIN_SUSPEND_REC_FAILURE. The guest will then take no further action in response to the request. The intent here is for the user to be presented with a warning message derived from the reason field indicating that the suspend operation failed and (if applicable) that a particular recovery operation failed. If a recovery operation failed, the user would then inspect the guest domain and take any action required to cleanup after failed recovery.

30.17.4.5. Resume and Post-suspend

After the guest has successfully suspended itself by calling into the hypervisor, the domain manager performs an out-of-band operation to resume the domain.

After the guest has been resumed, the guest will perform any post-suspend processing which results in either success or failure. In the event of success, the guest will send a response message with *result* value DOMAIN_SUSPEND_POST_SUCCESS and a *req_num* value equal to the request *req_num*. The guest will then take no further action in response to the request and the suspend operation will have completed successfully, leaving the domain in a normal state. In the event of failure, the guest will send a response message with *result* value DOMAIN_SUSPEND_POST_FAILURE, a *req_num* value equal to the request *req_num*, and optionally a NUL-terminated *reason* string describing the reason for the failure. The guest will then take no further action in response to the request. At this point, the guest is operational, subject to how the failing post-suspend processing leaves the guest. The intent is for the user to be presented with a warning message, derived from the reason string, indicating that the domain was resumed, but that a particular post-suspend operation failed. The user would then inspect the guest domain and take any action required to cleanup after failed post-suspend processing.

30.17.5. Message Sequences

The following sections describe the possible message sequences.

30.17.5.1. Sequence 1 (failure)

```
-> Request:  (req_num:n, type: Invalid)
<- Response: (req_num:n, result:DOMAIN_SUSPEND_INVALID_MSG,
                  rec_result: DOMAIN_SUSPEND_REC_SUCCESS, reason:0)
```

30.17.5.2. Sequence 2 (failure)

```
-> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
<- Response: (req_num:n, result:DOMAIN_SUSPEND_INPROGRESS
                  rec_result: DOMAIN_SUSPEND_REC_SUCCESS, reason: 0)
```

30.17.5.3. Sequence 3 (failure)

```
-> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
<- Response: (req_num:n, result:DOMAIN_PRE_SUSPEND_FAILURE,
                rec_result: DOMAIN_SUSPEND_REC_SUCCESS,
                reason: [optional string])
```

30.17.5.4. Sequence 4 (failure)

```
-> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
<- Response: (req_num:n, result:DOMAIN_PRE_SUSPEND_FAILURE,
                rec_result: DOMAIN_SUSPEND_REC_FAILURE,
                reason: [optional string])
```

30.17.5.5. Sequence 5 (failure)

```
 -> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
 <- Response: (req_num:n, result:DOMAIN_SUSPEND_PRE_SUCCESS, reason: 0)
 <- Response: (req_num:n, result:DOMAIN_SUSPEND_FAILURE,
                 rec_result: DOMAIN_SUSPEND_REC_SUCCESS,
                 reason: [optional string])
```

30.17.5.6. Sequence 6 (failure)

```
-> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
<- Response: (req_num:n, result:DOMAIN_SUSPEND_PRE_SUCCESS, reason: 0)
<- Response: (req_num:n, result:DOMAIN_SUSPEND_FAILURE,
                rec_result: DOMAIN_SUSPEND_REC_FAILURE,
                reason: [optional string])
```

30.17.5.7. Sequence 7 (suspend success, post-suspend failure)

```
-> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
<- Response: (req_num:n, result:DOMAIN_SUSPEND_PRE_SUCCESS, reason: 0)
-- [suspend and resume occurs]
<- Response: (req_num:n, result:DOMAIN_SUSPEND_POST_FAILURE,
                rec_result: DOMAIN_SUSPEND_REC_SUCCESS,
                reason: [optional string])
```

30.17.5.8. Sequence 8 (success)

```
-> Request:  (req_num:n, type:DOMAIN_SUSPEND_SUSPEND)
<- Response: (req_num:n, result:DOMAIN_SUSPEND_PRE_SUCCESS, reason: 0)
-- [suspend and resume occurs]
<- Response: (req_num:n, result:DOMAIN_SUSPEND_POST_SUCCESS,
                rec_result: DOMAIN_SUSPEND_REC_SUCCESS,
                reason: 0)
```

Chapter 31. Diagnostic services

Guests may be allowed to invoke their own diagnostic code with hypervisor privileges. Such code is often used for bring-up, verification, and error injection purposes. It is expected that these services will only be used internally at Sun.

The hypervisor will disallow the services defined here unless explicitly configured otherwise. The diagnostic code may destabilize the entire platform, including other guests, as it runs without restriction. There are no air bags.

31.1. API calls

31.1.1. diag_ra2pa

trap#	FAST_TRAP
function#	DIAG_RA2PA
arg0	*ra*
ret0	*status*
ret1	*pa*

Translates a guest's real address into the underlying platform's physical address.

This is required to specify the diagnostic code invoked by the diag_hexec call as well as to pass pointers to guest data structures to the diagnostic code.

It is not guaranteed that the entire span of an object is physically-contiguous simply because it is contiguous in the real address space. Care must be taken when using code or data larger than the smallest page size the platform supports.

31.1.1.1. Errors

ENOACCESS	Guest not permitted to invoke diag_ra2pa
ENORADDR	Invalid real address *ra*

31.1.2. diag_hexec

trap#	FAST_TRAP
function#	DIAG_HEXEC
arg0	*codepa*
ret0	*status*

Invokes diagnostic code located at physical address *codepa* at the next higher trap level. The diagnostic code executes in a hyperprivileged environment.

The caller may specify other arguments and the invoked code may return other return values. The code is run in the hyperprivileged, processor-specific environment of the underlying hardware. Arguments that are guest pointers (virtual or real) will have to be converted to physical addresses using diag_ra2pa prior to invoking this service.

The diagnostic code is expected to execute a SPARC v9 done instruction to return to the caller. How or if the code returns cannot be enforced by the hypervisor.

If the guest is not permitted to make this call then *ret0* will contain ENOACCESS. Otherwise the invoked diagnostic code is expected to set *ret0* appropriately.

31.1.2.1. Errors

ENOACCESS	Guest not permitted to invoke `diag_hexec`

Appendix A. Number Registry

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

A.1. API Groups

The definitions of the API groupings for the versioning API (Chapter 11, *API versioning*) are as follows:

Table A.1. API Groups

Group	Group#	Definition
Common	0x000	sun4v platform
Common	0x001	Core APIs
Common	0x002	Interrupt APIs
Common	0x003	Guest Soft State
	0x004	*Reserved*
Technology	0x100	PCI
Technology	0x101	Logical Domain Channels
Technology	0x102	Service Channels
Technology	0x103	Niagara Crypto Services
Technology	0x104	Niagara Random Number Generator
Common	0x105	Parallel Boot Services
Common	0x106	Sun4v FMA Support Services
Technology	0x107	Trusted Platform Module (TPM) Services
Technology	0x108	Reserved for PCI Static Direct I/O and PCIe IOV
Technology	0x109	Reserved for PCI Static Direct I/O Trivial Error Model
Common	0x110	Reboot Data Services
Performance Measurement	0x200	UltraSPARC-T1 performance counters
Performance Measurement	0x201	Fire PCI performance counters
Performance Measurement	0x202	UltraSPARC-T2 performance counters
Performance Measurement	0x203	UltraSPARC-T2 PIU performance counters
Technology	0x204	UltraSPARC-T2 NIU Services
Performance Measurement	0x205	UltraSPARC-T2+ performance counters
Performance Measurement	0x209	UltraSPARC-T3 performance counters
Performance Measurement	0x20a	UltraSPARC-T3 IOS performance counters
Common	0x20e	Global De-Map
Test & Development	0x300	Platform-specific optional test interfaces

A.2. Hyper-fast Trap numbers

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

The use of unassigned trap numbers result in EBADTRAP being returned in %o0 as described in section 2.3.

A.3. FAST_TRAP Function numbers

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

The use of unassigned function numbers used for fast-traps result in EBADTRAP being returned in %o0 as described in section 2.3.

A.4. 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
: See Section 11.1.1, “api_set_version”

API_PUTCHAR
: an alias for FAST_TRAP function CONS_PUTCHAR, see Section 18.1.2, “cons_putchar”

API_EXIT
: an alias for FAST_TRAP function MACH_EXIT, see Section 11.1.2, “api_get_version”

API_GET_VERSION
: defined in Section 11.1.2, “api_get_version”

A.5. Summary of trap and function numbers

Table A.2. Trap and Function numbers

Trap#	Func#	Group#	Vers#	Name	Reference
0x80	—	—	—	FAST_TRAP	
0x83	—	0x001	1.0	MMU_MAP_ADDR	Section 14.8.6
0x84	—	0x001	1.0	MMU_UNMAP_ADDR	Section 14.8.8
0x85	—	0x001	1.0	TTRACE_ADDENTRY	Section 21.3.5
0xff	—	—	—	CORE_TRAP	
0x80	0x00	0x001	1.0	MACH_EXIT	Section 12.1.1
0x80	0x01	0x001	1.0	MACH_DESC	Section 12.1.2
0x80	0x02	0x001	1.0	MACH_SIR	Section 12.1.3
0x80	0x05	0x001	1.1	MACH_SET_WATCHDOG	Section 12.1.4
0x80	0x10	0x001	1.0	CPU_START	Section 13.2.1
0x80	0x11	0x001	1.1	CPU_STOP	Section 13.2.2
0x80	0x12	0x001	1.0	CPU_YIELD	Section 13.2.5
0x80	0x13			*reserved*	*reserved*
0x80	0x14	0x001	1.0	CPU_QCONF	Section 13.2.6
0x80	0x15	0x001	1.0	CPU_QINFO	Section 13.2.7
0x80	0x16	0x001	1.0	CPU_MYID	Section 13.2.9

Trap#	Func#	Group#	Vers#	Name	Reference
0x80	0x17	0x001	1.0	CPU_STATE	Section 13.2.10
0x80	0x18	0x001	1.0	CPU_SET_RTBA	Section 13.2.3
0x80	0x19	0x001	1.0	CPU_GET_RTBA	Section 13.2.4
0x80	0x20	0x001	1.0	MMU_TSB_CTX0	Section 14.8.1
0x80	0x21	0x001	1.0	MMU_TSB_CTXNON0	Section 14.8.2
0x80	0x22	0x001	1.0	MMU_DEMAP_PAGE	Section 14.8.3
0x80	0x23	0x001	1.0	MMU_DEMAP_CTX	Section 14.8.4
0x80	0x24	0x001	1.0	MMU_DEMAP_ALL	Section 14.8.5
0x80	0x25	0x001	1.0	MMU_MAP_PERM_ADDR	Section 14.8.7
0x80	0x26	0x001	1.0	MMU_FAULT_AREA_CONF	Section 14.8.10
0x80	0x27	0x001	1.0	MMU_ENABLE	Section 14.8.11
0x80	0x28	0x001	1.0	MMU_UNMAP_PERM_ADDR	Section 14.8.9
0x80	0x29	0x001	1.0	MMU_TSB_CTX0_INFO	Section 14.8.12
0x80	0x2a	0x001	1.0	MMU_TSB_CTXNON0_INFO	Section 14.8.13
0x80	0x2b	0x001	1.0	MMU_FAULT_AREA_INFO	Section 14.8.14
0x80	0x31	0x001	1.0	MEM_SCRUB	Section 15.1.1
0x80	0x32	0x001	1.0	MEM_SYNC	Section 15.1.2
0x80	0x42	0x001	1.0	CPU_MONDO_SEND	Section 13.2.8
0x80	0x50	0x001	1.0	TOD_GET	Section 17.1.1
0x80	0x51	0x001	1.0	TOD_SET	Section 17.1.2
0x80	0x60	0x001	1.0	CONS_GETCHAR	Section 18.1.1
0x80	0x61	0x001	1.0	CONS_PUTCHAR	Section 18.1.2
0x80	0x62	0x001	1.1	CONS_READ	Section 18.1.3
0x80	0x63	0x001	1.1	CONS_WRITE	Section 18.1.4
0x80	0x70	0x001	1.0	SOFT_STATE_SET	Section 19.1.1
0x80	0x71	0x001	1.0	SOFT_STATE_GET	Section 19.1.2
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	Section 21.3.1
0x80	0x91	0x001	1.0	TTRACE_BUF_INFO	Section 21.3.2
0x80	0x92	0x001	1.0	TTRACE_ENABLE	Section 21.3.3
0x80	0x93	0x001	1.0	TTRACE_FREEZE	Section 21.3.4
0x80	0x94	0x001	1.0	DUMP_BUF_UPDATE	Section 20.1.1
0x80	0x95	0x001	1.0	DUMP_BUF_INFO	Section 20.1.2
0x80	0xa0	0x002	1.0	INTR_DEVINO2SYSINO	Section 16.3.1

Trap#	Func#	Group#	Vers#	Name	Reference
0x80	0xa1	0x002	1.0	INTR_GETENABLED	Section 16.3.2
0x80	0xa2	0x002	1.0	INTR_SETENABLED	Section 16.3.3
0x80	0xa3	0x002	1.0	INTR_GETSTATE	Section 16.3.4
0x80	0xa4	0x002	1.0	INTR_SETSTATE	Section 16.3.5
0x80	0xa5	0x002	1.0	INTR_GETTARGET	Section 16.3.6
0x80	0xa6	0x002	1.0	INTR_SETTARGET	Section 16.3.7
0x80	0xa7	0x002	2.0	VINTR_GETCOOKIE	Section 16.2.1
0x80	0xa8	0x002	2.0	VINTR_SETCOOKIE	Section 16.2.2
0x80	0xa9	0x002	2.0	VINTR_GETENABLED	Section 16.2.3
0x80	0xaa	0x002	2.0	VINTR_SETENABLED	Section 16.2.4
0x80	0xab	0x002	2.0	VINTR_GETSTATE	Section 16.2.5
0x80	0xac	0x002	2.0	VINTR_SETSTATE	Section 16.2.6
0x80	0xad	0x002	2.0	VINTR_GETTARGET	Section 16.2.7
0x80	0xae	0x002	2.0	VINTR_SETTARGET	Section 16.2.8
0x80	0xb0	0x100	1.1	PCI_IOMMU_MAP	Section 23.4.1
0x80	0xb1	0x100	1.0	PCI_IOMMU_DEMAP	Section 23.4.2
0x80	0xb2	0x100	1.1	PCI_IOMMU_GETMAP	Section 23.4.3
0x80	0xb3	0x100	1.0	PCI_IOMMU_GETBYPASS	Section 23.4.4
0x80	0xb4	0x100	1.0	PCI_CONFIG_GET	Section 23.4.5
0x80	0xb5	0x100	1.0	PCI_CONFIG_PUT	Section 23.4.6
0x80	0xb6	0x100	1.0	PCI_PEEK	Section 23.4.7
0x80	0xb7	0x100	1.0	PCI_POKE	Section 23.4.8
0x80	0xb8	0x100	1.0	PCI_DMA_SYNC	Section 23.4.9
0x80	0xc0	0x100	1.0	PCI_MSIQ_CONF	Section 24.4.1
0x80	0xc1	0x100	1.0	PCI_MSIQ_INFO	Section 24.4.2
0x80	0xc2	0x100	1.0	PCI_MSIQ_GETVALID	Section 24.4.3
0x80	0xc3	0x100	1.0	PCI_MSIQ_SETVALID	Section 24.4.4
0x80	0xc4	0x100	1.0	PCI_MSIQ_GETSTATE	Section 24.4.5
0x80	0xc5	0x100	1.0	PCI_MSIQ_SETSTATE	Section 24.4.6
0x80	0xc6	0x100	1.0	PCI_MSIQ_GETHEAD	Section 24.4.7
0x80	0xc7	0x100	1.0	PCI_MSIQ_SETHEAD	Section 24.4.8
0x80	0xc8	0x100	1.0	PCI_MSIQ_GETTAIL	Section 24.4.9
0x80	0xc9	0x100	1.0	PCI_MSI_GETVALID	Section 24.4.10
0x80	0xca	0x100	1.0	PCI_MSI_SETVALID	Section 24.4.11
0x80	0xcb	0x100	1.0	PCI_MSI_GETMSIQ	Section 24.4.12
0x80	0xcc	0x100	1.0	PCI_MSI_SETMSIQ	Section 24.4.13
0x80	0xcd	0x100	1.0	PCI_MSI_GETSTATE	Section 24.4.14
0x80	0xce	0x100	1.0	PCI_MSI_SETSTATE	Section 24.4.15

Trap#	Func#	Group#	Vers#	Name	Reference
0x80	0xd0	0x100	1.0	PCI_MSG_GETMSIQ	Section 24.4.16
0x80	0xd1	0x100	1.0	PCI_MSG_SETMSIQ	Section 24.4.17
0x80	0xd2	0x100	1.0	PCI_MSG_GETVALID	Section 24.4.18
0x80	0xd3	0x100	1.0	PCI_MSG_SETVALID	Section 24.4.19
0x80	0xe0	0x101	1.0	LDC_TX_QCONF	Section 22.4.1
0x80	0xe1	0x101	1.0	LDC_TX_QINFO	Section 22.4.2
0x80	0xe2	0x101	1.0	LDC_TX_GET_STATE	Section 22.4.3
0x80	0xe3	0x101	1.0	LDC_TX_SET_QTAIL	Section 22.4.4
0x80	0xe4	0x101	1.0	LDC_RX_QCONF	Section 22.4.5
0x80	0xe5	0x101	1.0	LDC_RX_QINFO	Section 22.4.6
0x80	0xe6	0x101	1.0	LDC_RX_GET_STATE	Section 22.4.7
0x80	0xe7	0x101	1.0	LDC_RX_SET_QHEAD	Section 22.4.8
0x80	0xea	0x101	1.0	LDC_SET_MAP_TABLE	Section 22.5.1
0x80	0xeb	0x101	1.0	LDC_GET_MAP_TABLE	Section 22.5.2
0x80	0xec	0x101	1.0	LDC_COPY	Section 22.5.3
0x80	0xed	0x101	1.1	LDC_MAPIN	Section 22.5.4
0x80	0xee	0x101	1.1	LDC_UNMAP	Section 22.5.5
0x80	0xef	0x101	1.1	LDC_REVOKE	Section 22.5.6
0x80	0xf8	0x108	1.0	PCI_IOV_ROOT_CONFIGURED	Section 23.5.2.1
0x80	0xf9	0x108	1.0	PCI_REAL_CONFIG_GET	Section 23.5.2.2
0x80	0xfa	0x108	1.0	PCI_REAL_CONFIG_PUT	Section 23.5.2.3
0x80	0xff	0x109	1.0	PCI_ERROR_SEND	Section 23.5.2.4
0x80	0x100	0x200	1.0	NIAGARA_GET_PERFREG	Section 27.1.1
0x80	0x101	0x200	1.0	NIAGARA_SET_PERFREG	Section 27.1.2
0x80	0x102	0x200	1.0	NIAGARA_MMUSTAT_CONF	Section 27.2.2
0x80	0x103	0x200	1.0	NIAGARA_MMUSTAT_INFO	Section 27.2.3
0x80	0x104	0x202	1.0	NIAGARA2_GET_PERFREG	Section 27.4.4
0x80	0x105	0x202	1.0	NIAGARA2_SET_PERFREG	Section 27.4.5
0x80	0x106	0x205	1.0	VFALLS_GET_PERFREG	Section 27.5.9
0x80	0x107	0x205	1.0	VFALLS_GET_PERFREG	Section 27.5.8
0x80	0x111	0x103	2.0	NCS_QCONF	Section 25.2.3
0x80	0x112	0x103	2.0	NCS_QINFO	Section 25.2.4
0x80	0x113	0x103	2.0	NCS_GETHEAD	Section 25.2.5
0x80	0x114	0x103	2.0	NCS_GETTAIL	Section 25.2.7
0x80	0x115	0x103	2.0	NCS_SETTAIL	Section 25.2.8
0x80	0x116	0x103	2.0	NCS_QHANDLE_TO_DEVINO	Section 25.2.9
0x80	0x117	0x103	2.0	NCS_SETHEAD_MARKER	Section 25.2.6
0x80	0x118	0x103	2.1	NCS_ULQCONF	Section 25.2.10

Trap#	Func#	Group#	Vers#	Name	Reference
0x80	0x120	0x201	1.0	FIRE_GET_PERFREG	Section 27.3.2
0x80	0x121	0x201	1.0	FIRE_SET_PERFREG	Section 27.3.3
0x80	0x122	0x209	1.0	KT_GET_PERFREG	Section 27.6.5
0x80	0x123	0x209	1.0	KT_SET_PERFREG	Section 27.6.6
0x80	0x130	0x104	1.0	RNG_GET_DIAG_CONTROL	Section 25.1.12.1
0x80	0x131	0x104	1.0	RNG_CTL_READ	Section 25.1.12.2
			2.0	RNG_CTL_READ	Section 25.1.9
0x80	0x132	0x104	1.0	RNG_CTL_WRITE	Section 25.1.12.3
			2.0	RNG_CTL_WRITE	Section 25.1.10
0x80	0x133	0x104	1.0	RNG_DATA_READ_DIAG	Section 25.1.12.4
			2.0	RNG_DATA_READ_DIAG	Section 25.1.11
0x80	0x134	0x104	1.0, 2.0	RNG_DATA_READ	Section 25.1.8
0x80	0x140	0x203	1.0	N2PIU_GET_PERF_REG	Section 27.4.7
0x80	0x141	0x203	1.0	N2PIU_SET_PERF_REG	Section 27.4.8
0x80	0x142	0x204	1.0	N2NIU_RX_LP_SET	Section 26.4.1
			2.0	N2NIU_RX_LP_SET	Section 26.6.1
0x80	0x143	0x204	1.0	N2NIU_RX_LP_GET	Section 26.4.2
			2.0	N2NIU_RX_LP_GET	Section 26.6.2
0x80	0x144	0x204	1.0	N2NIU_TX_LP_SET	Section 26.4.3
			2.0	N2NIU_TX_LP_SET	Section 26.6.1
0x80	0x145	0x204	1.0	N2NIU_TX_LP_GET	Section 26.4.4
			2.0	N2NIU_TX_LP_GET	Section 26.6.2
0x80	0x146	0x204	1.0	N2NIU_VR_ASSIGN	Section 26.5.1.1
			2.0	N2NIU_VR_ASSIGN	Section 26.6.3.1
0x80	0x147	0x204	1.0, 2.0	N2NIU_VR_UNASSIGN	Section 26.5.1.2
0x80	0x148	0x204	1.0, 2.0	N2NIU_VR_GETINFO	Section 26.5.1.3
0x80	0x149	0x204	1.0, 2.0	N2NIU_VR_RX_DMA_ASSIGN	Section 26.5.2.1
0x80	0x14a	0x204	1.0, 2.0	N2NIU_VR_RX_DMA_UNASSIGN	Section 26.5.2.2
0x80	0x14b	0x204	1.0, 2.0	N2NIU_VR_TX_DMA_ASSIGN	Section 26.5.2.1
0x80	0x14c	0x204	1.0, 2.0	N2NIU_VR_TX_DMA_UNASSIGN	Section 26.5.2.2
0x80	0x14d	0x204	1.0, 2.0	N2NIU_VR_GET_RX_MAP	Section 26.5.2.3
0x80	0x14e	0x204	1.0, 2.0	N2NIU_VR_GET_TX_MAP	Section 26.5.2.3
0x80	0x150	0x204	1.0, 2.0	N2NIU_VRRX_SET_INO	Section 26.5.2.4
0x80	0x151	0x204	1.0, 2.0	N2NIU_VRTX_SET_INO	Section 26.5.2.4
0x80	0x152	0x204	1.0, 2.0	N2NIU_VRRX_GET_INFO	Section 26.5.2.5
0x80	0x153	0x204	1.0, 2.0	N2NIU_VRTX_GET_INFO	Section 26.5.2.5
0x80	0x154	0x204	1.0, 2.0	N2NIU_VRRX_LP_SET	Section 26.5.2.6
0x80	0x155	0x204	1.0, 2.0	N2NIU_VRRX_LP_GET	Section 26.5.2.7

Trap#	Func#	Group#	Vers#	Name	Reference
0x80	0x156	0x204	1.0, 2.0	N2NIU_VRTX_LP_SET	Section 26.5.2.6
0x80	0x157	0x204	1.0, 2.0	N2NIU_VRTX_LP_GET	Section 26.5.2.7
0x80	0x158	0x204	1.0, 2.0	N2NIU_VRRX_PARAM_GET	Section 26.5.3.1
0x80	0x159	0x204	1.0, 2.0	N2NIU_VRRX_PARAM_SET	Section 26.5.3.2
0x80	0x15a	0x204	1.0, 2.0	N2NIU_VRTX_PARAM_GET	Section 26.5.3.1
0x80	0x15b	0x204	1.0, 2.0	N2NIU_VRTX_PARAM_SET	Section 26.5.3.2
0x80	0x165	0x20a	1.0	KT_IOS_GET_PERFREG	Section 27.6.8
0x80	0x166	0x20a	1.0	KT_IOS_SET_PERFREG	Section 27.6.9
0x80	0x170	0x105	1.0	MACH_PRI	Section 12.1.6
0x80	0x171	0x110	1.0	MACH_REBOOT_DATA_GET	Section 12.1.9
0x80	0x172	0x110	1.0	MACH_REBOOT_DATA_SET	Section 12.1.8
0x80	0x176	0x107	1.0	TPM_GET	Section 25.3.2.1
0x80	0x177	0x107	1.0	TPM_PUT	Section 25.3.2.2
0x80	0x178	0x105	1.0	MACH_VARS	Section 12.1.7
0x80	0x181	0x1	1.2	MACH_SUSPEND	Section 12.1.5
0x80	0x182	0x1	1.2	CPU_TICK_NPT	Section 13.2.11
0x80	0x183	0x1	1.2	CPU_STICK_NPT	Section 13.2.12
0x80	0x1a2	0x20e	1.0	MMU_GLOBAL_DEMAP_PAGE	Section 14.8.15
0x80	0x1a3	0x20e	1.0	MMU_GLOBAL_DEMAP_CTX	Section 14.8.16
0x80	0x1a4	0x20e	1.0	MMU_GLOBAL_DEMAP_ALL	Section 14.8.17
0x80	0x1a5	0x20e	1.0	MMU_GLOBAL_DEMAP_STATUS	Section 14.8.18
0x80	0x200	0x300	1.0	DIAG_RA2PA	Section 31.1.1
0x80	0x201	0x300	1.0	DIAG_HEXEC	Section 31.1.2
0xff	0x00	—	—	API_SET_VERSION	Section 11.1.1
0xff	0x01	—	—	CONS_PUTCHAR	Section 18.1.2
0xff	0x02	—	—	MACH_EXIT	Section 12.1.1
0xff	0x03	—	—	API_GET_VERSION	Section 11.1.2

A.6. 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.

Table A.3. Error codes

Value	Mnemonic	Description
0	EOK	Successful return
1	ENOCPU	Invalid CPU ID
2	ENORADDR	Invalid real address
3	ENOINTR	Invalid interrupt ID

Value	Mnemonic	Description
4	EBADPGSZ	Invalid page size 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 the error state
13	ENOTSUPPORTED	Function is not supported
14	ENOMAP	No mapping found
15	ETOOMANY	Too many items specified or limit reached
16	ECHANNEL	Invalid LDC channel
17	EBUSY	Operation failed because resource is otherwise busy
18	EPENDING	A long-running operation was started and the corresponding status function needs to be called

Appendix B. Domain Service Registry

This table lists the capabilities described in this document, and which need to be added to a Domain Services registry.

Table B.1. Domain Services

Service	Description	Reference
agent-device	Guest device information	
agent-dio	Direct I/O agent	
agent-system	Guest system information	
asr	Primary ASR management service	
asr-backup	Secondary ASR management service	
domain-panic	Request a panic	Section 30.7
domain-shutdown	Request a graceful shutdown	Section 30.6
domain-suspend	Domain Suspend service	Section 30.17
dr-cpu	Dynamic Reconfiguration for CPUs	Section 30.8
dr-mem	Dynamic Reconfiguration for memory	Section 30.9
dr-vio	Dynamic Reconfiguration for virtual I/O	Section 30.10
dr-crypto-mau	Dynamic Reconfiguration for MAU devices	Section 30.11
dr-crypto-cwq	Dynamic Reconfiguration for CWQ devices	Section 30.11
fma-cpu-service	CPU Online/Oflline service for fault management	
fma-io-domain-service	I/O service for fault management	
fma-mem-service	Memory retire service for fault management	
fma-pri-service	Domain Manager service for retrieving the PRI	
fma-phys-cpu-service	Physical CPU Online/Offline service for fault management	
fma-phys-mem-service	Physical Memory retire/resurrect service for fault management	
fw-progress-state	Primary FW Progress State	
keystore	Primary keystore for WANboot service	Section 30.13
keystore-backup	Secondary keystore for WANboot service	Section 30.13
md-update	Notification of MD updates	Section 30.5
pm-rm	Power-Management Resource Manager	
pri-update	Notification of PRI updates	Section 30.14
snmp	SNMP service	Section 30.16
system-info	Guest System Information	Section 30.15
var-config	Primary LDom variable management	Section 30.12
var-config-backup	Secondary LDom variable management	Section 30.12

Appendix C. Physical Resource Inventory

C.1. Introduction

This is the specification for the Physical Resource Inventory (PRI). The PRI is intended to contain physical information about a system. Much of this information had previously been contained in the Machine Description (MD) provided to the guest. With the advent of Logical Domains (LDoms), it becomes essential to have a clean separation between the physical and virtual views of a system. To achieve this, much of the physical information is no longer part of the MD within a guest domain. This provides a strictly virtual view of the system within an LDom.

However, besides the System Controller (SC), other management entities (e.g. FMA, the LDom Manager) still require a physical view of the system. This is now provided by the PRI. The PRI includes the physical information which is no longer available in the MD (needed by FMA to perform diagnosis). It also includes additional information necessary for the LDom Manager to configure and manage individual logical domains, including acting as a template for the LDom Manager to construct MDs for the various logical domains it is managing.

The PRI is provided by the System Controller (SC) to the domain. The PRI contains physical information about system resources in a format identical to that used by the guest MD. The names for the nodes and properties used to access that information are defined to be platform independent, as are many of the values. Some of the values may be platform-dependent and may require additional documentation to parse them.

The PRI may be updated at anytime on the SC depending on the state of its resources. Protocols and transports may be agreed upon with the domain for delivering the PRI to it so that it may use the information for activities such as configuring LDOM guest domains or diagnosing error data.

One of the primary consumers of the PRI on the domain is the LDOM Manager. It uses the PRI to generate guest MDs by extracting information from it that serves as a template for new domains. The PRI may contain any of the nodes and properties that may be found in a guest MD on a system. For this reason, this case imports all past and future cases that specify new nodes or properties for the guest MD that may be used on a system. Since future cases cannot be explicitly imported, any new cases that arise should specify that they apply to both the guest MD and the PRI.

In addition to containing the information needed by the LDOM Manager for creating guest MDs, the PRI also supplies the information it needs to generate the hypervisor data needed to boot a set of configured guest domains. This hypervisor data may include information about either the SC or other firmware components that it needs to know about. Some items in this specification in this category include the LDC endpoints and host prom information.

Another of the primary consumers of the PRI are FMA modules such as diagnosis engines. There is a requirement to provide additional information in the PRI so that FMA may make diagnoses and act on them through fault, repair or logging actions. To enable this, the PRI contains system-wide information that may not be available in the guest MD in the domain on which FMA modules are running. As an example, FMA modules running in the control domain (the single domain running the LDOM Manager) may handle ereports generated for resources owned by other guest domains in the system. It may access the PRI to get information it needs to diagnose and act on data in the ereports. For more background, see the FWARC case 2006/141 FMA Domain Services.

In addition to the requirements spelled out in the FMA Domain Services case, there is a requirement to add additional information to the PRI so that FMA may generate reports with more specific details about

components that have been diagnosed faulty. Some of this information is available in a FRUID present on the component or on another component that is proxying the information. It is the responsibility of the SC to derive this information for components present on a system and to populate the relevant property values in the PRI.

The PRI also represents how components are contained within other components in a hierarchical fashion in the system. Each component can be identified as a FRU or not based on a property. The parent component that contains each component can also be found by following an arc property back to it. Based on this, for any component in a system, its closest parent component that is a FRU can be found so that it may be replaced if faulty. There should be only one parent component for each component, but a component may have multiple children components if they are all physically contained by it. The idea of a component physically containing another component is meant to indicate how they may be removed or replaced and is determined on a platform-specific basis.

FMA also uses the hierarchy of the components node to build the FM topology in Solaris. The hierarchy encoded into the PRI directly translates to the resulting topology in Solaris. In addition to hierarchy, new properties have been added to guide how topology is enumerated in the OS.

C.2. Root Node

Name:	`root`
Category:	core
Required subordinates:	`components` (Section C.3, “Components Node”)

This node is the top-most node of the PRI.

C.2.1. PRI version property

Name	Tag	Required?
`pri-version`	`PROP_STR`	yes

This is the version string for the PRI. Format of string is “<x>.<y>”, where “<x>” denotes the decimal major version number and “<y>” denotes the decimal minor version number. In this context, *major* and *minor* version numbers connote incompatible and compatible changes respectively, as defined in the API versioning interface specified by FWARC 2005/499 and documented in the API versioning chapter (Chapter 11, *API versioning*). The currently defined version is "1.0".

C.3. Components Node

Name:	`components`
Category:	core
Required subordinates:	`component` (Section C.4, “Component Node”)

This node is the parent of all component nodes and has a back arc to the root node.

C.3.1. Power Management (PM) versioning property

Name	Tag	Required?
`pm-version`	`PROP_STR`	yes

The version string for the content of the Power Management related information in the PRI. The currently defined version is “1.0”.

C.4. Component Node

Name:	`component`
Category:	resource required
Required subordinates:	
Optional subordinates:	`component` (Section C.4, “Component Node”)

The component nodes represent physical entities in the system in order to provide component information and system topology to the domain.

C.4.1. type Property

Name	Tag	Required?
`type`	`PROP_STR`	yes

This property contains the type of the component. Only types that have been submitted with an ARC case should be present.

The currently acceptable values for the `type` property include:

`product`	The product.
`chassis`	A system chassis.
`systemboard`	A system board.
`dimm`	A single memory DIMM.
`processor`	A physical processor that is logically a single agent on the system interconnect fabric. It may contain several processor cores or strands.
`core`	A physical processor core.
`sp`	A service processor.
`strand`	A physical strand in a processor.
`mem-board`	A physical memory board.
`cpu-board`	A physical cpu board.
`io`	An I/O device such as a switch, bridge, slot, or leaf device.

C.4.1.1. Type-specific property requirements

The following requirements hold for component nodes of these types:

For processors that have multiple cores, the `fwd` property of the processor type node should link to a core type node. For core type nodes, the `fwd` property should link to a strand type node. If there are multiple component nodes for multiple processors in the system, this can be used to determine which processor a core is on, and which core a strand is on.

Any type of node may have an `id` property. The `id` property must be unique within a set of nodes with the same `topo-hc-name` value as well as immutable across system reset events. In addition, for root

complex IO nodes (i.e. nodes whose `topo-hc-name` value is “hostbridge”), the `id` property is further constrained to have whatever platform-specific binding is required for this value to denote a unique and persistent root complex ID. This property is required for nodes that do *not* set the `topo-skip` property. Grandfather clause: `id` is *always* required for strand, mem-board, and cpu-board nodes.

Any type of node may have a `nac` property. A `nac` property is required for component nodes that are FRUs (`fru` property = 0x1).

An io type node may have a `path` property, as described below. For io nodes describing XAUI cards, the `path` property is required.

Any type of node may set the `topo-hc-name` property. This property is required for nodes that do *not* set the `topo-skip` property. The property is also required, irrespective of the `topo-skip` setting, for io nodes that describe XAUI cards.

Any type of node may set the `topo-skip` property. For io nodes describing XAUI cards, this property *must* be set to a non-zero value.

Nodes representing a root complex must use a `topo-hc-name` of "hostbridge" and set the `path` property. The previous requirements for setting the `cfg-handle` property for a node representing a root complex is now deprecated.

Nodes representing a root port must use a `topo-hc-name` of "pciexrc". This apparent misnomer is intentional, as it follows the convention established on the x64 platforms.

The product and chassis type nodes must have serial_number properties.

Any type node that is a FRU (`fru` = 0x1) are required to include the `serial_number`, `part_number`, `rev_number`, and `dash_number` properties.

The components node *must* provide a `topo-hc-name` hierarchy of hostbridge/pciexrc. No intervening nodes are allowed. Failure to comply will result in a Solaris-side FM topology that does not support PCIE fabric diagnosis. The node hierarchy above the hostbridge is arbitrary per platform.

For platforms wishing to employ the sun4v platform-independent SPARC cpu diagnosis engine, *one* of the following nodes must be present: processor, core, or strand. One, some or all can be used. The most logical hierarchy is processor/core/strand.

For platforms wishing to employ the sun4v platform-independent SPARC memory diagnosis engine, the following `topo-hc-name` hierarchy *must* be provided in the components node: chip/memory-buffer *OR* memory-controller/memory-buffer. No intervening nodes are allowed. The node hierarchy above or below is arbitrary per platform.

For platforms wishing to employ diagnosis of a faulted service processor, a `topo-hc-name` hierarchy of chassis/sp *must* be provided in the components node. Furthermore, the node describing the “sp” must be tagged as a FRU.

For platforms wishing to employ diagnosis of internal hard disks, io nodes describing hard drive bays (e.g. `nac` = “HDD#”) must specify a `topo-hc-name` of “bay” and the `id` property. The `id` property must match the physical identification (i.e. HDD0 is id 0, HDD1 is id 1, etc.).

C.4.2. nac Property

Name	Tag	Required?
nac	PROP_STR	yes

This property contains the NAC for the component, as described in the system nomenclature document for the system. It may appear in any type of component node. It is only required for nodes that are FRUs (`fru` = 0x1).

C.4.3. fru Property

Name	Tag	Required?
`fru`	`PROP_VAL`	no

This property is present and has a value of 1 if the component is a FRU.

C.4.4. serial_number Property

Name	Tag	Required?
`serial_number`	`PROP_STR`	no

This property contains the component serial number contained in the FRUID. It is required for “product” and “chassis” type nodes and nodes that are that are FRUs (`fru` = 0x1).

C.4.5. part_number Property

Name	Tag	Required?
`part_number`	`PROP_STR`	yes

This property contains the component part number contained in the FRUID. It is only required for nodes that are FRUs (`fru` = 0x1).

C.4.6. rev_number Property

Name	Tag	Required?
`rev_number`	`PROP_STR`	yes

This property contains the component rev number contained in the FRUID. It is only required for nodes that are FRUs (`fru` = 0x1).

C.4.7. dash_number Property

Name	Tag	Required?
`dash_number`	`PROP_STR`	yes

This property contains the component dash number contained in the FRUID. It is only required for nodes that are FRUs (`fru` = 0x1).

C.4.8. id Property

Name	Tag	Required?
`id`	`PROP_STR`	yes

This property contains the physical id (physical with respect to the system) of a resource in the system. It is required for all nodes, *except* those that set `topo-skip` to a non-zero value. The `id` property must be unique within a set of nodes with the same `topo-hc-name` value. In other words, no two nodes where `topo-hc-name`=“dimm” can have the same `id` value, but a node with `topo-hc-name`=“dimm” and a node with `topo-hc-name`=“cpu” can have the same `id` value.

C.4.9. path Property

Name	Tag	Required?
path	PROP_STR	yes

This property contains the canonical path of an I/O device, composed of its full device path with device names removed. It is required on nodes where the `topo-hc-name` property is set to “hostbridge”, “pciexrc”, “bay” or “xaui”.

It is possible to find the FRU parent of an I/O device by performing a search beginning with its parent and continuing through its ancestry until reaching a component node with a `fru` property with value of 1.

C.4.10. label Property

Name	Tag	Required?
label	PROP_STR	no

This property is currently only defined to appear in dimm type component nodes. It will contain the “J” number that is silk-screened on the board next to the dimm slot.

C.4.11. name Property

Name	Tag	Required?
name	PROP_STR	no

This property is a human readable string describing the component. Examples are “CPU Chip 0”, “CPU Chip 0 Core 0”, and “Strand 1”.

C.4.12. pm_resource Property

Name	Tag	Required?
pm_resource	PROP_STR	no

This property indicates the type of resource that will stop carrying load if the resource has been transitioned to a state that yields zero performance. Possible values are: “CPU”, “IO”, etc. It is consumed by the PM Engine.

C.4.13. pm_states Property

Name	Tag	Required?
pm_states	PROP_DATA	no

This property encodes multiple string tuples. Within each tuple are two elements; the first element describes the performance value at the power state which is described by the second element. Performance values are in units of 0.1%.

Examples are,

For “strand” type nodes,

```
pm_states = {
        "0 Parked",
```

```
        "1000 Unparked"
};
```

For “core” type nodes,

```
pm_states = {
        "0 Disabled",
        "1000 Enabled"
};
```

For “processor” type nodes,

```
pm_states = {
        "125 One-Eighth_Speed",
        "250 One-Fourth_Speed",
        "375 Three-Eighths_Speed",
        "500 Half_Speed",
        "625 Five-Eighths_Speed",
        "750 Three-Fourths_Speed",
        "875 Seven-Eighths_Speed",
        "1000 Full_Speed"
};
```

C.4.14. pm_cookie Property

Name	Tag	Required?
pm_cookie	PROP_STR	no

This property encodes a string. The string is a resource identifier that is consumed by the PM Engine. Each string consists of a space-delineated set of tokens, which taken together describe an instance of a particular resource type. For pm-version 1.0, each string is a tuple describing a resource type and a type-specific identifier. Examples are “F 0”, “C 3”, and “S 12”.

C.4.15. pm_dependency Property

Name	Tag	Required?
pm_dependency	PROP_STR	no

This property describes the power dependencies between the component and its children. It is consumed by the PM Engine. An example value is “StrictParental Performance”.

C.4.16. pm_coordination Property

Name	Tag	Required?
pm_coordination	PROP_STR	no

This property describes the power dependencies between the component and its peers. It is consumed by the PM Engine. An example value is “NeedOne Defragment”.

C.4.17. pm_mapping Property

Name	Tag	Required?
pm_mapping	PROP_STR	no

This property describes the mapping from the component to the identity in the LDom manager's name space of the resource upon which a reconfiguration operation may be required prior to transitioning the component to zero performance state, or after transitioning the component out of zero performance state.

An example value is “cpu 7”, which means:

- cpu known to the LDom manager as (physical) cpu 7
- when the PM Engine sets the component into or out of the zero performance state, reconfiguration can be effected by asking the LDom manager to reconfigure cpu 7 prior to or after the power state transition.

C.4.18. topo-hc-name Property

Name	Tag	Required?
topo-hc-name	PROP_STR	yes

This property contains the FM libtopo hc canonical name string for a node. It is used by Solaris enumerators when instantiating a topology. The value of `topo-hc-name` must be an approved name per FMA's hc scheme ([fmahc]). It is required for all nodes that do not use/set `topo-skip`. It is also required on io nodes describing XAUI cards, irrespective of that node's `topo-skip` value.

C.4.19. topo-skip Property

Name	Tag	Required?
topo-skip	PROP_VAL	sometimes

When this property is present and has a non-zero integer value, the sun4v platform-independent enumerator will *not* create FM topology nodes for this node or any of its children. This property can be used on any type of node. This property is required and must be a non-zero value for io nodes describing XAUI cards.

C.4.20. assignable-path Property

Name	Tag	Required?
assignable-path	PROP_STR	no

For a PCI-e assignable device (one where ownership can be transferred to another guest domain), this defines the path of the assignable unit. This path is similar to, but not identical to, the `path` property defined in Section C.4.9, “path Property”.

C.4.21. pm_power Property

Name	Tag	Required?
pm_power	PROP_DATA	no

This property encodes multiple string tuples. Within each tuple are two elements: The first element describes the performance value for a Processor power state. The second element is an encoded string of Cores powered on and the power consumed by those Cores at the given Processor power state.

For example, using the first entry below: At full chip speed, 3 cores enabled consumes 500 Watts, 2 cores enabled consumes 400 Watts, and 1 core enabled consumes 50 Watts.

```
pm_power = {
        "500 3.475;2.225;1.20",
        "1000 3.500;2.400;1.50"
};
```

C.5. Firmware Node

Name:	`firmware`
Category:	core
Required subordinates:	`read_only_memory` (Section C.6, “Read_Only_Memory Node”)

This node contains information describing firmware constraints needed by the LDOM Manager for configuring guest domains.

C.5.1. max_guests Property

Name	Tag	Required?
`max_guests`	`PROP_VAL`	yes

This property describes the maximum number of guests that the firmware supports.

C.5.2. max_hv_ldcs Property

Name	Tag	Required?
`max_hv_ldcs`	`PROP_VAL`	yes

This property describes the maximum number of hypervisor LDC endpoints that the hypervisor supports.

C.5.3. max_guest_ldcs Property

Name	Tag	Required?
`max_guest_ldcs`	`PROP_VAL`	yes

This property describes the maximum number of guest LDC endpoints that the hypervisor supports.

C.5.4. max_guest_dependencies Property

Name	Tag	Required?
`max_guest_dependencies`	`PROP_VAL`	no

This property describes the maximum number of guest dependencies (per guest) that the firmware supports. If not present, should be treated as 0.

C.5.5. directio_capability Property

Name	Tag	Required?
`directio_capability`	`PROP_VAL`	no

This property indicates that the platform and this version of firmware are capable of supporting Static Direct I/O PCI-e virtualization.

C.6. Read_Only_Memory Node

Name:	`read_only_memory`
Category:	core
Required subordinates:	`rom_img` (Section C.7, “Rom_Img Node”)

This node contains information about the contents of read-only memory on the system, such as the host or system PROM.

C.6.1. name Property

Name	Tag	Required?
`name`	`PROP_STR`	yes

This property contains a human readable string for identifying the read-only memory that this node represents. For example, “System PROM” may be used to indicate that this is the PROM for the system firmware.

C.6.2. base Property

Name	Tag	Required?
`base`	`PROP_VAL`	yes

This property contains the base address of the read-only memory in the system address space.

C.6.3. size Property

Name	Tag	Required?
`size`	`PROP_VAL`	yes

This property contains the size of the read-only memory in bytes.

C.7. Rom_Img Node

Name:	`rom_img`
Category:	core
Required subordinates:	

This node contains information about a firmware component in read-only memory.

C.7.1. name Property

Name	Tag	Required?
`name`	`PROP_STR`	yes

This property contains a human readable string for identifying this firmware image in the read-only memory. For example, “Openboot” may be used to indicate that this image is the OpenBoot image used to boot the guest.

C.7.2. offset Property

Name	Tag	Required?
offset	PROP_VAL	yes

This property contains the offset into read-only memory of this firmware image.

C.7.3. size Property

Name	Tag	Required?
size	PROP_VAL	yes

This property contains the size of the firmware image in bytes.

C.7.4. alignment Property

Name	Tag	Required?
alignment	PROP_VAL	no

This property contains any memory alignment requirements of the firmware image, for placing that image in memory so that it runs and boots successfully.

C.7.5. min_allocation Property

Name	Tag	Required?
min_allocation	PROP_VAL	no

This property contains any minimum memory allocation requirements of the firmware image.

C.7.6. guest_use Property

Name	Tag	Required?
guest_use	PROP_VAL	no

This property indicates that this firmware image is suitable for use as the start-up image for a guest.

C.8. Ldc_Endpoints Node

Name:	ldc_endpoints
Category:	core
Required subordinates:	ldc_endpoint (Section C.9, “Ldc_Endpoint Node”)

This node aggregates fwd arc links to all the ldc_endpoint nodes needed by the LDOM Manager for generating the information hypervisor needs to configure its internal data structures to route packets between LDC channel endpoints.

C.9. Ldc_Endpoint Node

Name:	ldc_endpoint
Category:	core

Required subordinates:

This node contains the information hypervisor needs to configure its internal data structures to route packets between LDC channel endpoints.

C.9.1. resource_id Property

Name	Tag	Required?
`resource_id`	`PROP_VAL`	yes

This property contains a unique id for each ldc_endpoint node.

C.9.2. target_type Property

Name	Tag	Required?
`target_type`	`PROP_VAL`	yes

This property contains a value indicating one of several types of targets that is on the other end of a pair of LDC endpoints.

Acceptable values are:

0	The target endpoint is a guest
1	The target endpoint is the hypervisor
2	The target endpoint is the system controller

C.9.3. channel Property

Name	Tag	Required?
`channel`	`PROP_VAL`	yes

This property contains the endpoint id for the channel endpoint this node represents. The channel endpoint may be owned by a guest, hypervisor or the SP.

C.9.4. target_channel Property

Name	Tag	Required?
`target_channel`	`PROP_VAL`	yes

This property contains the endpoint id for the target endpoint on the other end of this channel. The target channel endpoint may be owned by a guest, hypervisor or the SP.

C.9.5. tx-ino and rx-ino Properties

Name	Tag	Required?
`tx-ino`	`PROP_VAL`	yes
`rx-ino`	`PROP_VAL`	yes

These properties contains the same values as the corresponding `tx-ino` and `rx-ino` properties in the `channel-endpoint` node in the guest MD. The `channel-endpoint` node has an `id` property value

matching this `ldc_endpoint` node `channel` property value. This enables the hypervisor to target interrupts to the guest LDC endpoint.

C.10. Memory Segments and related nodes

Name:	`memory-segments`
Category:	resource required
Required subordinates:	`memory-segment` (Section C.11, “Memory-Segment Node”)

Child of the root node with fwd arcs to the memory-segment nodes.

C.11. Memory-Segment Node

Name:	`memory-segment`
Category:	resource required
Required subordinates:	`memory-bank` (Section C.12, “Memory-Bank Node”)

Describes a contiguous memory address range. Its properties define that address range and they link to child nodes that specify criteria for locating a physical address in the memory segment to a set of one or more DIMMs that constitute a memory-bank.

A memory-segment node has the following properties.

C.11.1. base Property

Name	Tag	Required?
`base`	`PROP_VAL`	yes

The base physical address of the range represented by this memory segment.

C.11.2. size Property

Name	Tag	Required?
`size`	`PROP_VAL`	yes

The size of the address range represented by this memory segment.

C.12. Memory-Bank Node

Name:	`memory-bank`
Category:	resource required
Required subordinates:	`component` (Section C.4, “Component Node”)

Contains properties that describe the constraints for determining if a physical address is located on the set of one or more DIMMs that comprise this memory bank.

The `memory-bank` node has `fwd` arcs to component nodes with DIMM type properties. The DIMM type component nodes contain `nac` properties used to identify the DIMM. If an address belongs to this memory bank, it is located on one of the DIMM type component nodes that are linked to by this node.

C.12.1. size Property

Name	Tag	Required?
size	PROP_VAL	yes

The size of this memory bank.

C.12.2. mask Property

Name	Tag	Required?
mask	PROP_VAL	yes

The value of the mask property is logically and'd with a physical address and the result is compared with the value in the match property to determine if the physical address is in this memory-bank.

C.12.3. match Property

Name	Tag	Required?
match	PROP_VAL	yes

After the value of the mask property is and'd with a physical address, if the resultant value is equal to the value of the match property, the address is on one of the DIMMs in this memory bank.

C.13. IO Device node

Name:	iodevice
Category:	optional
Required subordinates:	
Optional subordinates:	iodevice (Section C.13, “IO Device node”), interrupt-map-entry (Section C.14, “Interrupt mapping node”),

Refer to Section 8.25.2, “I/O device node”, the PRI inherits properties from the machine description iodevice-node. Additional properties or modifications to inheritied properties are listed in this section.

C.13.1. Sun4v to PCI Express root nexus device

See Section 8.25.2.2.1, “Sun4v to PCI Express root nexus device” for inherited properties.

C.13.1.1. rcid Property

Name	Tag	Required?
rcid	PROP_VAL	yes

Root complex identifier. Monotonically assigned, the only requirement is that each PCI-e root complex have a unique identifier.

C.13.2. Generic PCI device properties

See Section 8.25.2.2.2, “Generic PCI device properties”.

C.13.3. PCI bridge type device properties

See Section 8.25.2.2.3, “PCI bridge device properties”.

C.13.3.1. chassis-location-name Property

Name	Tag	Required?
`chassis-location-name`	`PROP_STR`	no

This property, when present, contains a NAC name string matching a node in the “components” portion of the PRI graph. This property will exist in the device nodes where a FRU boundary has been crossed in the PCIE fabric. It will exist in the first device node entry pertaining to the FRU.

This property is only applicable to pcie-switch-upstream and pcie-switch-downstream device types. It is *only* for locations within a chassis. It does not apply for slot adapters or any sub-frus external to the chassis itself.

C.13.4. PCI slot type device properties

See Section 8.25.2.2.4, “PCI slot device properties”.

C.13.5. PCI network device properties

See Section 8.25.2.2.5, “PCI network device properties”.

C.13.6. PCI SCSI device properties

See Section 8.25.2.2.6, “PCI SCSI device properties”.

C.14. Interrupt mapping node

Name:	`interrupt-map-entry`
Category:	optional
Required subordinates:	
Optional subordinates:	

See Section 8.25.4, “Interrupt mapping node”.

C.15. Power-Management node

Name:	`power-management`
Category:	core
Required subordinates:	
Optional subordinates:	`memory-grouping` (Section C.16, “Memory-Grouping Node”)

This node is a child of root node and its children represent the system topology from a power management perspective. Devices like memory which can not directly be mapped to component nodes are defined here.

C.16. Memory-Grouping Node

Name:	`memory-grouping`

Category:	core
Required subordinates:	`memory-region` (Section C.17, “Memory-Region Node”),
Optional subordinates:	

This node represents memory as it is grouped to form a power manageable entity.

C.16.1. id Property

Name	Tag	Required?
`id`	`PROP_VAL`	yes

This property contains an id which uniquely identifies a memory grouping in the system.

C.16.2. name Property

Name	Tag	Required?
`name`	`PROP_STR`	yes

This property contains a human readable string to identify a memory grouping.

C.16.3. pm_resource Property

Name	Tag	Required?
`pm_resource`	`PROP_STR`	yes

This property indicates the type of resource that will stop carrying load if the resource has been transitioned to a state that yields zero performance. The string value of this property for memory grouping is “Memory”. It is consumed by the PM Engine.

C.16.4. pm_states Property

Name	Tag	Required?
`pm_states`	`PROP_DATA`	yes

This property encodes multiple string tuples. Within each tuple are two elements; the first element describes the performance value at the power state which is described by the second element. Performance values are in units of 0.1%.

Example:

```
pm_states = {
        "30 SlowExit",
        "1000 FastExit"
};
```

C.16.5. pm_cookie Property

Name	Tag	Required?
`pm_cookie`	`PROP_STR`	yes

This property encodes a string. The string is a resource identifier that is consumed by the PM Engine. Each string consists of a space-delineated set of tokens, which taken together describe an instance of a particular resource type. A memory grouping with `id` value `2` will be represented as string value of “M 2”

C.17. Memory-Region Node

Name:	`memory-region`
Category:	core
Required subordinates:	
Optional subordinates:	

This node represents a contiguous memory region within the parent `memory-grouping`.

C.17.1. id Property

Name	Tag	Required?
`id`	`PROP_VAL`	yes

This property contains an id which uniquely identifies a memory region in the system.

C.17.2. name Property

Name	Tag	Required?
`name`	`PROP_STR`	yes

This property contains a human readable string to identify the memory region and to which memory grouping this region belongs.

C.17.3. base Property

Name	Tag	Required?
`base`	`PROP_VAL`	yes

The base physical address represented by this memory region.

C.17.4. size Property

Name	Tag	Required?
`size`	`PROP_VAL`	yes

The size of this memory region in bytes.

Bibliography

[sparcv9] 0-13-825001-4. David L. Weaver and Tom Germond. Copyright © 1994 SPARC International, Inc.. PTR Prentice Hall. *The SPARC Architecture Manual, Version 9 [http://www.sparc.org/standards/SPARCV9.pdf]*.

[ua2005] 950-5553-12. Copyright © 2008 Sun Microsystems, Inc.. Sun Microsystems, Inc.. *UltraSPARC Architecture 2005 Specification [http://opensparc-t2.sunsource.net/specs/UA2007-current-draft-HP-EXT.pdf]*. 19 June, 2008. Hyperprivileged Edition.

[ua2005n1] 819-3404-04. Copyright © 2006 Sun Microsystems, Inc.. Sun Microsystems, Inc.. *UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005 [http://opensparc-t1.sunsource.net/specs/UST1-UASuppl-current-draft-HP-EXT.pdf]*. 17 March, 2006.

[ua2007] 950-5553-12. Copyright © 2008 Sun Microsystems, Inc.. Sun Microsystems, Inc.. *UltraSPARC Architecture 2007 Specification [http://opensparc-t2.sunsource.net/specs/UA2007-current-draft-HP-EXT.pdf]*. 19 June, 2008. Hyperprivileged Edition.

[ua2007n2] 950-5556-02. Copyright © 2006 Sun Microsystems, Inc.. Sun Microsystems, Inc.. *UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007 [http://opensparc-t2.sunsource.net/specs/UST2-UASuppl-current-draft-HP-EXT.pdf]*.

[ua2007n3] Copyright © 2009 Sun Microsystems, Inc.. Sun Microsystems, Inc.. *UltraSPARC {KT} Supplement to the UltraSPARC Architecture 2007 [https://systemsweb.sfbay.sun.com/kt/team/arch_specs/index.html]*.

[ua2009] 950-5554-13. Copyright © 2009 Sun Microsystems, Inc.. Sun Microsystems, Inc.. *UltraSPARC Architecture 2009 Specification*. 31 March, 2009. Hyperprivileged Edition.

[pcie2002] Copyright © 2002 PCI SIG. PCI SIG. *PCI Express Base Specification Revision 1.0 [http://www.pcisig.com/specifications/pciexpress/specifications/]*. 29 April, 2002.

[sun4vbind] Sun Microsystems, Inc.. "Sun4v Bus Binding to Open Firmware (FWARC 2005/111) [http://arc.opensolaris.org/caselog/FWARC/2005/111/]". 14 March, 2005.

[lddr] Sun Microsystems, Inc.. "Logical Domain Dynamic Reconfiguration Specification [http://cpubringup.sfbay.sun.com/twiki/pub/LDoms/ArchDesignPhase15/dr-design.pdf]". 21 February, 2006.

[efi] 731843-001. Intel Corp.. "Extensible Firmware Interface Specification [http://developer.intel.com/technology/efi/main_specification.htm]". 12 December, 2000.

[scsi3] INCITS T10. "SCSI Standards Architecture [http://www.t10.org/scsi-3.htm]".

[diskids] Sun Microsystems, Inc.. "PSARC 1995/352 Disk ID [http://sac.sfbay/Archives/CaseLog/arc/PSARC/1995/352/]". 10 April, 1996.

[tpm] Trusted Computing Group. "TPM Specification v1.2 [http://www.trustedcomputinggroup.org/resources/tpm_main_specification]". 26 October, 2006.

[tpmpc] Trusted Computing Group. "PC Client Work Group PC Specific Implementation Specification Version 1.1 [http://www.trustedcomputinggroup.org/resources/pc_client_work_group_pc_specific_implementation_specification_version_11]". 18 Auguest, 2003.

[ofintrmap] Open Firmware Working Group. "Open Firmware Recommended Practice: Interrupt Mapping (v0.9) [http://playground.sun.com/1275/practice/imap/imap0_9d.pdf]". 12 July, 1996.

[msiprops] Sun Microsystems, Inc.. “MSI Properties (FWARC 2005/030) [http://arc.opensolaris.org/caselog/FWARC/2005/030/materials/msi-props.txt]”. 18 January, 2005.

[vpcierrs] Sun Microsystems, Inc.. “Sun4v PCI Error Packet Definitions [http://pciexpress.sfbay.sun.com/fma/docs/sun4v-err.txt]”. 28 May, 2009.

[fmahc] Sun Microsystems, Inc.. “Solaris FMA hc scheme names [http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/fm/topo/libtopo/common/topo_hc.h]”.