Domain Services Specification

Revision 0.9.9

March 6, 2006

Please send comments & queries to:

ryan.maeda@sun.com or

eric.sharakan@sun.com

Table of Contents

1 Introduction

1.1 Background

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.

1.2 Domain services protocol

This document 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.

1.2.1 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 in figure 1.

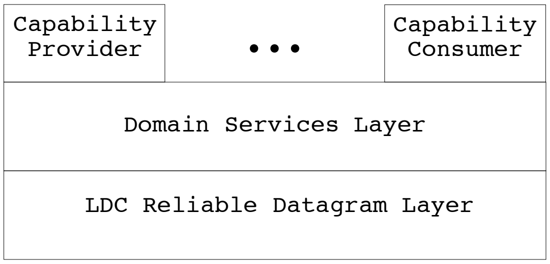

Figure 1: Communication Stack

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 in Figure 2.

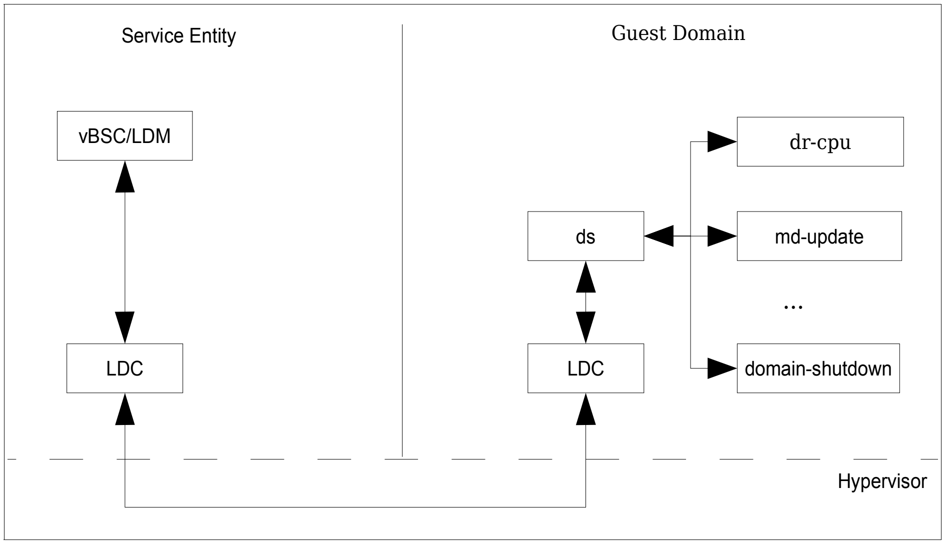

Figure 2: Domain Services Communication Path Example

2 Domain Services Protocol

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          /* MAXPATHLEN */
```

Note: For implementations in C, a data structure representation relevant for a big-endian architecture is also presented.

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

typedef struct {
       uint32_t     msg_type;
       uint32_t     payload_len;
} ds_hdr_t;
```

The data payload content is defined according to the `msg_type` field.

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.

2.3.1 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

```
typedef struct {
      uint16_t     major_vers;
      uint16_t     minor_vers;
} ds_init_req_t;
```

2.3.2 Initiation acknowledgment

msg_type:

```
DS_INIT_ACK         0x1
```

Payload:

Offset	Size	Field name	Description
0	2	minor_vers	Highest supported minor version

```
typedef struct {
      uint16_t     minor_vers;
} ds_init_ack_t;
```

2.3.3 Initiation negative acknowledgment

msg_type:

```
DS_INIT_NACK  0x2
```

Payload:

Offset	Size	Field name	Description
0	2	major_vers	Alternate supported major version

```
typedef struct {
      uint16_t     major_vers;
} ds_init_nack_t;
```

2.4 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 back-compatible changes; the two sides implicitly agree to use the lower of the two minor numbers exchanged, and the negotiation is successfully completed. This scheme is fully described as part of the versioning API, FWARC/2006/052.

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.

2.5 DS protocol version 1.0

2.5.1 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 2.5.4.1), 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.

2.5.2 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 the Machine Description Specification [md].

2.5.3 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

2.5.4 DS Message types defined for v.1.0 of the DS protocol

2.5.4.1 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	Var.	svc_id	Service name

```
typedef struct {
      uint64_t      svc_handle;
      uint16_t      major_vers;
      uint16_t      minor_vers;
      char          svc_id[];
} ds_reg_req_t;
```

2.5.4.2 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

```
typedef struct {
      uint64_t      svc_handle;
      uint16_t      minor_vers;
} ds_reg_ack_t;
```

2.5.4.3 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

typedef struct {
       uint64_t     svc_handle;
       uint64_t     result;
       uint16_t     major_vers;  /* DS_REG_VER_NACK only */
} ds_reg_nack_t;
```

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
```

2.5.4.4 Unregister Service

msg_type:

```
DS_UNREG            0x6
```

Payload:

```
Offset Size   Field name   Description
------ ----   ----------   -----------
0      8      svc_handle   Service handle to unregister

typedef struct {
       uint64_t     svc_handle;
} ds_unreg_req_t;
```

2.5.4.5 Unregister OK

msg_type:

```
DS_UNREG_ACK  0x7
```

Payload:

```
Offset Size   Field name   Description
------ ----   ----------   -----------
0      8      svc_handle   Service handle sent in DS_UNREG

typedef struct {
       uint64_t     svc_handle;
} ds_unreg_ack_t;
```

2.5.4.6 Unregister Failed

msg_type:

```
DS_UNREG_NACK 0x8
```

Payload:

Offset	Size	Field name	Description
0	8	svc_handle	Service handle sent in DS_UNREG

```
typedef struct {
      uint64_t     svc_handle;
} ds_unreg_nack_t;
```

2.5.4.7 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

```
typedef struct {
      uint64_t     svc_handle;
} ds_data_handle_t;
```

Note: The ds_data_handle_t header is defined so that when combined with the basic DS header the final payload delivered a service is aligned on a 64bit 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.

2.5.4.8 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

```
typedef struct {
      uint64_t     svc_handle;
      uint64_t     result;
} ds_data_nack_t;
```

A DS_NACK message can return the following result codes:

DS_INV_HDL	Service handle not valid
DS_TYPE_UNKNOWN	Unknown msg_type received

2.5.5 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.

2.5.6 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.

2.5.7 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.

3 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.

3.1 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.

3.1.1 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

3.1.2 MD Update Request

Payload:

Offset	Size	Field name	Description
0	8	req_num	Request number

```
typedef struct {
        uint64_t      req_num;
} md_update_req_t;
```

The req_num field is used to match up request & 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.

3.1.3 MD Update Response

Payload:

Offset	Size	Field name	Description
0	8	req_num	Request number
8	4	result	Result of operation

```
typedef struct {
      uint64_t     req_num;
      uint32_t     result;
} md_update_resp_t;

/* MD update result */
MD_UPDATE_SUCCESS          0x0
MD_UPDATE_FAILURE          0x1
MD_UPDATE_INVALID_MSG      0x2
```

3.2 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”).

3.2.1 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

3.2.2 Domain Shutdown Request

Offset	Size	Field name	Description
0	8	req_num	Request number
8	4	ms_delay	ms. to delay

```
typedef struct {
      uint64_t     req_num;
      uint32_t     ms_delay;
} domain_shutdown_req_t;
```

ms_delay specifies a time delay in milliseconds before initiation of the shutdown operation.

3.2.3 Domain Shutdown Response

Offset	Size	Field name	Description
0	8	req_num	Request number
8	4	result	Result of operation
12	Var.	reason	ASCII String (NUL terminated)

```
typedef struct {
      uint64_t      req_num;
      uint32_t      result;
      char          reason[];     /* Optional */
} domain_shutdown_resp_t;
```

reason is a NUL-terminated ASCII string.

```
/* Domain shutdown result */
DOMAIN_SHUTDOWN_SUCCESS           0x0
DOMAIN_SHUTDOWN_FAILURE           0x1
DOMAIN_SHUTDOWN_INVALID_MSG       0x2
```

3.3 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”).

3.3.1 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

3.3.2 Domain Panic Request

Offset	Size	Field name	Description
0	8	req_num	Request number

```
typedef struct {
      uint64_t      req_num;
} domain_panic_req_t;
```

3.3.3 Domain Panic Response

Offset	Size	Field name	Description
0	8	req_num	Request number
8	4	result	Result of operation
12	Var.	reason	ASCII String (NUL terminated)

```
typedef struct {
       uint64_t     req_num;
       uint32_t     result;
       char         reason[];     /* Optional */
} domain_panic_resp_t;
```

`reason` is a NUL terminated ASCII string.

```
/* Domain panic result */
DOMAIN_PANIC_SUCCESS              0x0
DOMAIN_PANIC_FAILURE              0x1
DOMAIN_PANIC_INVALID_MSG          0x2
```

3.4 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.

3.4.1 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.

3.4.2 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.

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

```
typedef struct {
       uint64_t     req_num;
       uint32_t     msg_type;
       uint32_t     num_records;
} dr_cpu_hdr_t;
```

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. An 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 identify either a request or a response to a request.

3.4.3 Message types

The following constants are defined for CPU DR domain service command identifier values:

Request message types:

Type	Value	ASCII	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	Definition
DR_CPU_OK	0x6f	'o'	Request completed OK
DR_CPU_ERROR	0x65	'e'	Request failed (not attempted)

3.4.3.1 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	Virtual CPU ID
4	4	id1	Virtual CPU ID
8	4	id2	Virtual CPU ID
			... etc.

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.

3.4.3.2 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.

3.4.3.3 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 HV by means of the CPU_START hypervisor API (FWARC 2005/116). Further steps required to reach the configured state is guest operating system specific. See [dr] for details on the Solaris specific implementation of the configure request.

3.4.3.4 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 HV from the guest by means of the CPU_STOP hypervisor API (FWARC 2005/116). Further steps required to reach the unconfigured state is guest operating system specific. See [dr] 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.

3.4.3.5 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.

3.4.3.6 CPU_STATUS

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.

3.4.4 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 relative to the first byte of the overall CPU DR message header described in section 3.4.2. (i.e. the `req_num` field described in section 3.4.2 is byte 0 of the offset). 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 start of CPU DR response packet

```
typedef struct {
      uint32_t     cpuid;
      uint32_t     result;
      uint32_t     status;
      uint32_t     string_off;
} dr_cpu_stat_t;
```

3.4.4.1 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
DR_CPU_RES_CPU_NOT_RESPONDING	0x3	CPU was not responding
DR_CPU_RES_NOT_IN_MD	0x4	CPU not defined in 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.

3.4.4.2 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 MD
DR_CPU_STAT_UNCONFIGURED	0x1	CPU ID exists in MD, but CPU is not configured for use by guest
DR_CPU_STAT_CONFIGURED	0x2	CPU is configured for use by the guest.

3.4.4.3 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.

3.4.5 CPU DR Error response

The message type DR_CPU_ERROR is returned as a response to a malformed request message. No additional payload is provided with this message type.

3.5 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. The LDOM variable store is described in detail in FWARC 2006/086.

3.5.1 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 LDOM variable management
“var-config-backup”	Secondary LDOM variable management

3.5.2 Message Header

Offset	Size	Field name	Description
0	4	cmd	Command

```
typedef struct {
      uint32_t     cmd;
} var_config_hdr_t;
```

3.5.2.1 Message types

The following constants are defined for Variable Configuration domain service command identifier values:

Type	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

3.5.3 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	Var.	name	Name of the variable to set
Var.	Var.	value	Value of variable

```
typedef struct {
        char          name[];
        char          value[];
} var_config_set_req_t;
```

3.5.4 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	Var.	name	Name of the variable to delete

```
typedef struct {
        char          name[];
} var_config_delete_req_t;
```

3.5.5 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

```
typedef struct {
        uint32_t      result;
} var_config_resp_t;
```

3.5.5.1 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
VAR_CONFIG_NO_SPACE	0x1	Variable Store 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

Appendix A: Capability Table

This table lists the capabilities described in this document, and which need to be added to a Domain Services registry.

Service ID	Description
`md-update`	Notification of MD updates
`domain-shutdown`	Request graceful shutdown
`domain-panic`	Request a panic
`dr-cpu`	Dynamic Reconfiguration for Virtual CPUs
`var-config`	Primary LDOM variable management
`var-config-backup`	Secondary LDOM variable management

Appendix B: References

[md] Machine Description Specification

FWARC/2005/115

[dr] Logical Domain Dynamic Reconfiguration Specification

http://cpubringup.sfbay.sun.com/twiki/pub/LDoms/ArchDesignPhase15/dr-design.pdf