SND 3.0
Interface and Implementation Guide

Document Number: IS000xx,REV00
Author: Peter Dunlap
Date: June 21, 1999

1.0 SND Architectural Overview 5
1.1 Changes in SND 3.0 5
1.2 Major Regions of SND 5
1.2.1 Personality Region 5
1.2.2 Zone Region 6
1.2.2.1 OIM - OS Interface Module 7
1.2.2.2 NIF- Network InterFace 7
1.2.2.3 SIF - Storage InterFace 8
1.2.2.4 BDM - Buffer Descriptor Manager 8
1.2.2.5 OSM - Operating system Services Module 8
1.2.2.6 Bus Module 8
1.2.3 Core Region 8
1.2.4 Bridge Region 8
1.3 Other SND Concepts 9
1.3.1 Personality Physical Drivers 9
1.3.2 Personality Virtual Drivers 9
1.3.3 Personality Service Drivers 9
1.3.4 Intermodule communications 9
1.3.5 Open Firmware Naming Convention 9
1.4 Basic Operations. 10
1.4.1 Initialization handshaking 10
1.4.1.1 PPD’s with associated OS interfaces 11
1.4.1.2 PPD’s without associated OS interfaces 12
1.4.1.3 PVD’s with associated OS interfaces 13
1.4.1.4 PVD’s without associated OS interfaces 13
1.4.1.5 PSD’s 13
1.4.2 Interrupt Handling 13
1.4.3 Packet Transmission 14
1.4.4 Packet Reception 15
1.4.5 SCSI Commands 15
2.0 Personality Implementation Guide 17
2.1 Level 1, Level2, Level3 functions 17
2.2 MP Issues 17
2.2.1 Region Based Locking 17
2.2.2 Fine Grained Locking 18
2.3 TX Interrupts 18
2.4 TX Flow Control 19
2.4.1 Return Value 19
2.4.2 tx_stop 19
2.4.3 tx_start 19
2.4.4 Interrupt Considerations for Flow Control 20
3.0 Zone Implementation Guide 39
3.1 SND Phases of Operation 39
3.2 Suggested Steps to Implement Zone 40
3.3 Relationship of BUS, OIM, NIF and SIF 40
3.4 Determining whether a device is a PPD, PVD or PSD 41
3.5 Order of initialization for PPD, PVD, PSD 41
3.6 Open Firmware Naming and Stable Storage 41
3.7 Using the Ifcon 41
3.8 MP Synchronization 42

3.9 Implementing the BDM 42
3.9.1 map vs. copy 42
3.9.2 How to map 43
3.9.3 When to map 43
3.9.4 SCSI commands in the BDM 43
3.9.5 Handling Receive Network Packets 43
3.10 Individual Function Implementations 44
3.10.1 OIM, NIF, SIF 45
3.10.1.1 oim_load_start 45
3.10.1.2 oim_stop_unload 45
3.10.1.3 oim_add_pers 45
3.10.1.4 oim_remove_pers 45
3.10.1.5 oim_add_bus 45
3.10.1.6 oim_remove_bus 45
3.10.1.7 oim_add_card_instance 45
3.10.1.8 oim_remove_card_instance 45
3.10.1.9 oim_ss_pvd_probe 45
3.10.1.10 oim_ss_get_pvd_child 45
3.10.1.11 oim_ss_check_os_if 45
3.10.1.12 nif_add_card_instance 45
3.10.1.13 nif_remove_card_instance 45
3.10.1.14 sif_add_card_instance 45
3.10.1.15 sif_remove_card_instance 45
3.10.2 BDM 45
3.10.2.1 bdm_connect 45
3.10.2.2 bdm_disconnect 45
3.10.2.3 bdm_tx_buf_alloc 45
3.10.2.4 bdm_tx_buf_sync 45
3.10.2.5 bdm_tx_buf_complete 45
3.10.2.6 bdm_tx_buf_free 46
3.10.2.7 bdm_rx_buf_alloc 46
3.10.2.8 bdm_rx_buf_peek 46
3.10.2.9 bdm_rx_buf_sync 46
3.10.2.10 bdm_rx_buf_free 46
3.10.2.11 bdm_del_buf_hdr 46
3.10.2.12 bdm_net_pkt_prepare 46
3.10.2.13 bdm_scsi_cmd_prepare 46
3.10.2.14 bdm_net_rx_giveaway 46
3.10.3 OSM 46
3.10.3.1 osm_i_malloc 46
3.10.3.2 osm_i_free 46
3.10.3.3 osm_i_timeout 46
3.10.3.4 osm_i_untimeout 46
3.10.3.5 osm_i_printstr 46
3.10.3.6 osm_logmsg 46
3.10.3.7 osm_i_createlock 46
3.10.3.8 OSM_I_LOCK 46
3.10.3.9 OSM_I_UNLOCK 46
3.10.4 BUS 46
3.10.4.1 <bus>_dma_init 46
3.10.4.2 <bus>_dma_handle_alloc 46
3.10.4.3 <bus>_dma_handle_free 47
3.10.4.4 <bus>_dma_map 47
3.10.4.5 <bus>_mm_alloc (bus_mm_zalloc will call <bus>_mm_alloc, then zero

the memory)47
3.10.4.6 <bus>_mm_free 47
3.10.4.7 <bus>_dma_unmap 47
3.10.4.8 <bus>_cache_sync 47
3.10.4.9 <bus>_isr_add 47
3.10.4.10<bus>_isr_remove 47
3.10.4.11<bus>_pio_map 47
3.10.4.12<bus>_pio_unmap 47
3.10.4.13<bus>_in_ubit8 47
3.10.4.14<bus>_in_ubit16 47
3.10.4.15<bus>_in_ubit32 47
3.10.4.16<bus>_in 47
3.10.4.17<bus>_out_ubit8 47
3.10.4.18<bus>_out_ubit16 47
3.10.4.19<bus>_out_ubit32 47
3.10.4.20<bus>_out 47
3.10.4.21<bus>_protect_region_from_dma 47
3.10.4.22<bus>_allow_dma_to_region 47
3.10.4.23<bus>_disconnect 47

4.0 Bus Bridge Implementation Guide 48

5.0 Core Interface Reference 49

5.1 General Description 49

6.0 Personality Interface Reference 143

6.1 General Description 143

6.5.1 Messages used by LAN personalities 157

7.0 Zone Interface Reference 173

8.0 Bridge Interface Reference 254

9.0 Using native code in personality modules 255

10.0 SND 2.x -> SND 3.x Delta 256

Relationship of SND Regions, OS, and Hardware 6
Example SND configuration showing Zone module relationships to Personality and Core modules. 7
SND Open Firmware Naming Syntax 10
Sequence of events for registering a personality module and binding it to a hardware adapter 11
Device Initialization State Machine 12
Example data flow for transmitted packet 14
Example data flow for received packet 15
Example data flow for SCSI command 16
16
SND Phases of operation 39
Data Flow Diagram for Device Distribution Among BUS, OIM, NIF, and SIF 40
Format of an SND Message Number 50

1.0 SND Architectural Overview

1.1 Changes in SND 3.0

There are a number of enhancements and changes to SND 2.x which have been incorporated into the SND 3.0 specification. A detailed list of changes is beyond the scope of this document (see 2.x->3.x delta document), but some of the major changes are as follows:

- Installable add-on modules, called Personality Virtual Drivers or PVD's, allow interfaces with no associated hardware. Examples are load balancing, fibre channel networking, and high availability extensions to our current products. See the section on personality virtual drivers for more information.
- Another new class of installable add-on modules, the Personality Service Driver or PSD, provides the capability to make a collection of code available to the other modules within the SND environment. This will be used initially for the FC link management code.
- Ifcon has been removed from the performance path.
- The ifcon_set_instance_attribs() interface has been replaced with a pure message passing mechanism
- Personality modules are now allowed to optionally handle their own MP synchronization.
- Personality modules can now provide native performance paths. In cases where the portable code does not have adequate performance, certain functions can be replaced with faster native code while retaining the portability of the remaining code.
- Most of the entry points in the card_mod_props structure have been eliminated, and the card_mod_props is now called pers_props (personality properties). Some functions are now performed by exchanging messages, while other entry points (such as send_pkt and isr) are explicitly provided within messages being exchanged. The remaining entry points in the pers_props structure are <Pers>_dev_connect and <Pers>_dev_disconnect.
- New entry points which must be provided by a Personality module when binding an interrupt handler are <Pers>_check_if_int_pending, <Pers>_disable_ints, and <Pers>_enable_ints. These are used in the NT environment, and also allow implementation of a Netware environment if necessary.
- Support for storage devices, including new storage oriented functional units in SND.
- Interface to BDM has been completely reworked. Buffers are now generic and can be used for storage or networking. packet_t is now called bdm_buf_t. In addition, operation specific structures for SCSI and networking have been added (scsi_cb_t and net_cb_t, respectively), which contain data which is specific to the ULP task.

1.2 Major Regions of SND

The SND environment is divided into regions which separate the code based on functionality and requirements for portability. Currently there are four major regions: Personality, Zone, Core and Bridge. The code for each region is located in the directory of the same name under the "SND_Source" directory. Figure 1.1 illustrates the relationship of the regions to the OS, the hardware, and to each other.

1.2.1 Personality Region

The personality region is made up of personality modules, each of which controls a particular adapter or chipset. A personality module may control more than one type of adapter in cases where the differences between adapters are minimal. An example of this would be the motofsi module, which controls both 4811 and 5511.

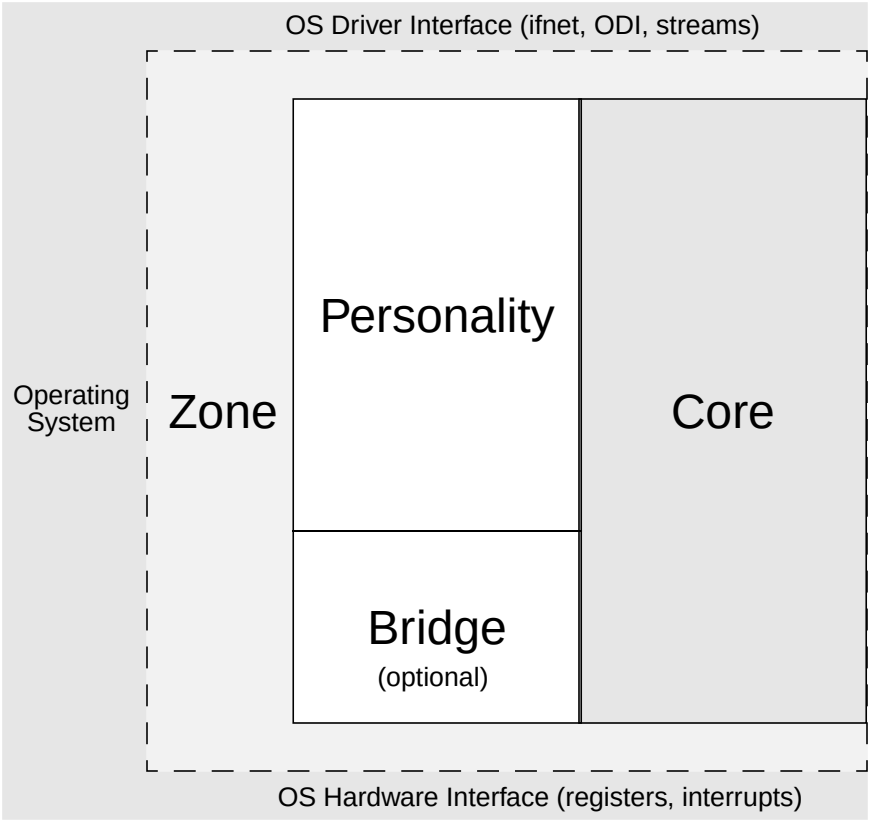


Figure 1.1 Relationship of SND Regions, OS, and Hardware

A personality module is responsible for discovering compatible devices in the system, initializing the device, sending packets or satisfying SCSI requests (depending on whether it's a storage or network module), and handling interrupts. One of the SND design goals is to make the personality module functions as simple and straightforward as possible, keeping more complex aspects of device management in the Zone where the necessary code only needs to be implemented once per supported operating system. Discovering compatible devices can mean simply recognizing a device ID passed down from the Zone if the device is in an EISA or PCI bus, or it could mean explicitly probing for the device in the case of VME.

1.2.2 Zone Region

OS and hardware specific issues are abstracted from the Personality, Core and Bridge regions by the Zone region. The Zone handles tasks such as device driver registration with the OS, mapping packets or SCSI requests to and from a personality module, accessing registers, synchronizing cache, and mapping memory. Because of the large number of tasks the Zone region is expected to handle, and because of the need to keep the code in the personality modules simple and straightforward, the Zone region is the largest collection of code, and the most difficult to implement. The Zone can be implemented either as one or more stand-alone pieces of object code which are somehow shared by the Personality and Bridge modules, or in can be statically linked to the Personality/Bridge modules in such a way that each Personality has its own copy of the code. See the Zone implementation guide for more information. Several functional units make up the Zone region, and are briefly described in the following sections. Figure 1.2 shows an example SND configuration which illustrates the functions of the Zone modules.

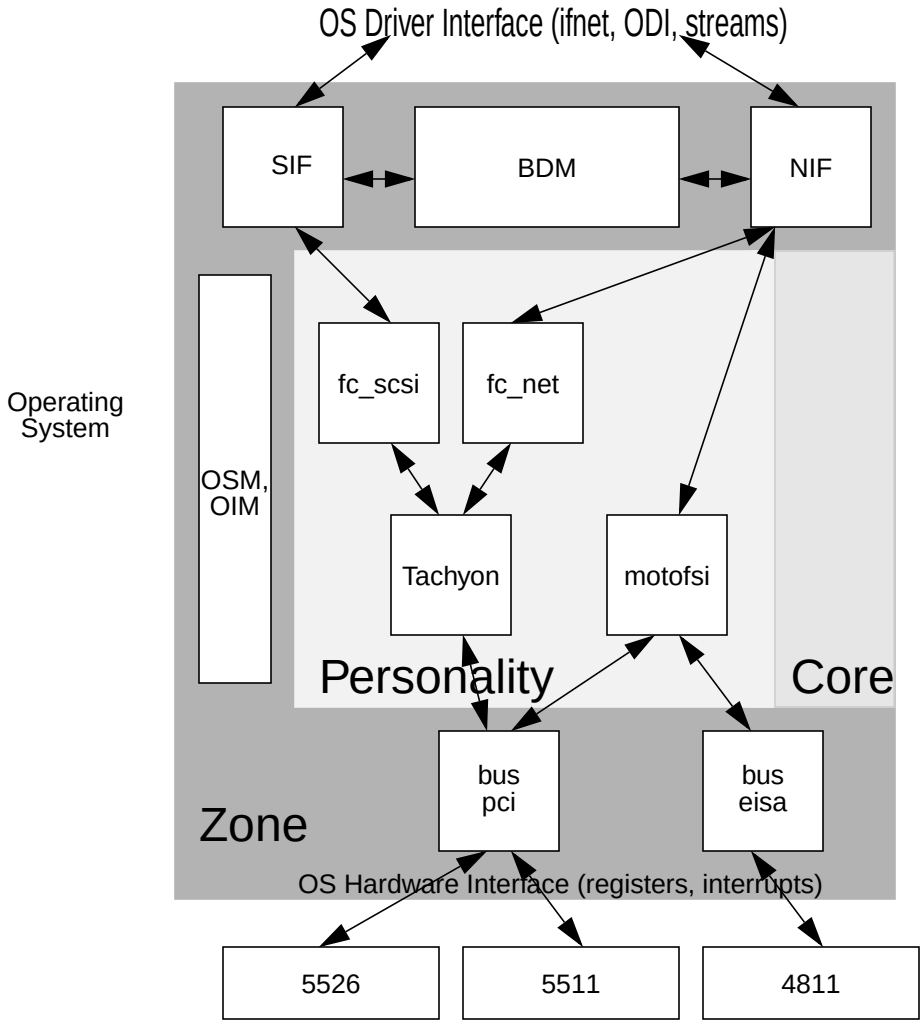


Figure 1.2 Example SND configuration showing Zone module relationships to Personality and Core modules.

1.2.2.1 OIM - OS Interface Module

At load time, buses and personalities register themselves with the OIM, and the OIM registers device drivers with the OS when appropriate. After a personality's dev_connect routine has been called, OIM forwards any discovered network adapter instances to the NIF and storage adapter instances to the SIF.

1.2.2.2 NIF- Network InterFace

The NIF is responsible for abstracting the operating system's I/O and network subsystems from the rest of the SND environment. This includes registering network devices with the operating system, handling transmit and receive packets in a manner which is consistent with the OS requirements, and satisfying other requirements of the OS for a network driver, such as IOCTLs and requests for information.

1.2.2.3 SIF - Storage InterFace

SIF is the storage equivalent of NIF. It interfaces to the storage subsystem of the OS, registering storage devices which have been discovered, and handling SCSI requests in the appropriate way for the OS. SIF is called by the OS and by the Personality region.

1.2.2.4 BDM - Buffer Descriptor Manager

Internally, SND uses a structure called bdm_buf_t to manage DMA buffers, including transmit and receive packets, and SCSI request buffers. The BDM provides a set of functions to manage bdm_buf_t structures, as well as translating OS buffer formats into scsi_cb_t or net_cb_t structures. NIF, SIF and the Personality region use BDM services.

1.2.2.5 OSM - Operating system Services Module

The OSM provides a standardized interface to common OS services, such as memory allocation, spinlocks, and timeouts. OSM services can be accessed from any SND module.

1.2.2.6 Bus Module

The bus module provides an abstraction of the hardware to the Card and Bridge region. Programmed I/O accesses (registers), mapping memory for DMA, and cache synchronization (except for structures controlled by NBDM or SBDM) are handled by the bus module. SND environments for most operating systems will actually have separate bus modules for each supported bus, since typically access to the different buses is not standardized. For simplicity's sake, in this document, "bus module" refers collectively to all the individual bus modules.

1.2.3 Core Region

The Core region contains definitions and code which are completely OS and hardware independent. Some of the code in Core is simply library code, including string manipulation functions, functions to help out with required SND tasks, and modules to provide services such as memory management. All the public interfaces to the various SND modules are defined in Core, along with standard PCI and EISA definitions.

The ifcon module is also contained within the Core region, which provides generic services for a particular interface, such as multicast address tracking, management of promiscuous mode state, and management of interface state.

1.2.4 Bridge Region

The Bridge Region is made up of bridge modules. A bridge module abstracts the details of a bus bridge from the personality modules associated with a device connected to the bridge. For example, a personality module for 5511 (PCI FDDI) might be controlling both a 5511 plugged into a PCI slot on a system, and a 5511 connected to a 6200 VME to PCI bus bridge board. The 6200 bridge module hides the details of the 6200 from the personality module. A bridge module is responsible for probing for devices attached to the bridge, providing I/O access to attached devices, controlling DMA engines on the bus bridge if it has any, handling interrupts from the bus bridge, and directing interrupts from the attached devices to the relevant personality modules. Bridge modules can talk to personality modules, bridge modules, and native bus modules. (insert diagram showing possible configurations).

1.3 Other SND Concepts

1.3.1 Personality Physical Drivers

Personality Physical Drivers or PPD's are personality modules which directly control hardware. In previous versions of SND, this was the only possible type of personality module. Only a PPD has an interface to a bus module and can use services from a bus module, unless a PVD or PSD driver ask for and receives a bus module handle from a child PPD. It is expected that only rare instances of PVD's and PSD's would require bus module services.

1.3.2 Personality Virtual Drivers

A Personality Virtual Driver or PVD does not control any hardware directly, but binds with another personality module and provides a virtual interface to the SIF or NIF. Multiple PVD's can be connected if desired. Examples of PVD's are fc_scsi and fc_net, both of which provide virtual interfaces using the tachyon personality module. Another example might be a high availability product which would create a virtual adapter from two physical adapters with redundant connections.

1.3.3 Personality Service Drivers

Some products require large portions of software which are specific to a network or storage technology, but not specific to a particular chipset. These software modules would be implemented as Personality Service Drivers or PSD's. Examples of software which could be implemented as a PSD are FDDI SMT, ATM signaling, and Fibre Channel FC-2.

1.3.4 Intermodule communications

SND 3.0 uses a message passing architecture for intermodule communications. Any module can register one or more message handlers with the ifcon message subsystem. Each message handler has a "handle" associated with it, which is usually a pointer to a control structure for the module the handler relates to. For example, a PPD or PVD would register its message handler with a pointer to its pers_dev_t structure as the handle. Messages are directed to a particular module either by the handle, or by a name which is also registered with the message handler.

The message subsystem is a simple low performance interface. It is expected that performance oriented data will be transferred via function interfaces, which could be established using the message interface.

1.3.5 Open Firmware Naming Convention

Each operating system has its own convention for naming devices. A portable environment like SND needs an internal naming convention which is consistent across operating systems. The open firmware naming convention provides this function, patterned after the Solaris naming. Figure 1.3 shows the syntax for open firmware naming. Figure 1.4 shows some examples.

A device's open firmware name up to <dtype> (see figure 1.3) is created during the dev_connect process. Each bus creates its portions of the name within its bus_t structure before calling its child device (bus bridge or personality). The personality then calls snd_build_OF_path() which creates the path string. Upon return from the dev_connect entry point, the OIM, NIF or SIF should then append the unique device name to the end of the string.

/<native_bus>{<bus#>}{@<nloc_id>}/{<bus_bridge>@<bloc_id>/}<dtype>/<udname>

<native_bus> = Name of native bus

<bus#> = Bus # of native bus (if it can be determined)

<nloc_id#> = Location of device in native bus. Could be slot number, device number or base address. Must be consistent across multiple boots of OS.

<bus_bridge> = Name of bus bridge

<bloc_id#> = Location of device in bus bridge. Whatever is appropriate for that bus bridge.

<dtype> = Type of device (4811, 5511, 5526, etc)

<udname> = Unique device name. Can be generated from bus#/nloc_id/bloc_id, or could be an OS device name.

Examples:

5511 on PCI bus#1 devid#6, OS network device is called ifi4

/pci1@6/5511/ifi4

4524 in PMC site 1 on 6200 bridge, mapped at 0xffc03000, OS net device is ife2

/vme@0xffc03000/6200@1/4524/ife2

4824 in EISA slot 4, OS net device is ife1

/eisa@4/4824/ife1

Figure 1.3 SND Open Firmware Naming Syntax

1.4 Basic Operations.

This section provides a general description of how some fundamental tasks are carried out with the SND 3.0 architecture.

1.4.1 Initialization handshaking

When the SND environment is started, the load entry points for all the configurable modules must be called (native bus modules, bus bridge modules, personality modules, etc). The load entry point will perform the necessary steps to allow communication with the module. For example, personality modules need to register themselves with OIM using oim_add_pers().

Calling the load entry points and device binding are handled by the OIM. After all the load entry points have been called, OIM knows about all the personality modules. Initialization now enters into the dev_connect phase, where physical devices installed in the system need to be matched up with their corresponding personality modules. For each physical device found in the system on a particular bus, its device information should be passed to every personality module which can support devices of that bus type. “Device information” in this context means either a device ID number or a device ID string. For buses which do not have the concept of device ID’s such as VME, the personality modules for that bus type should be called with NULL

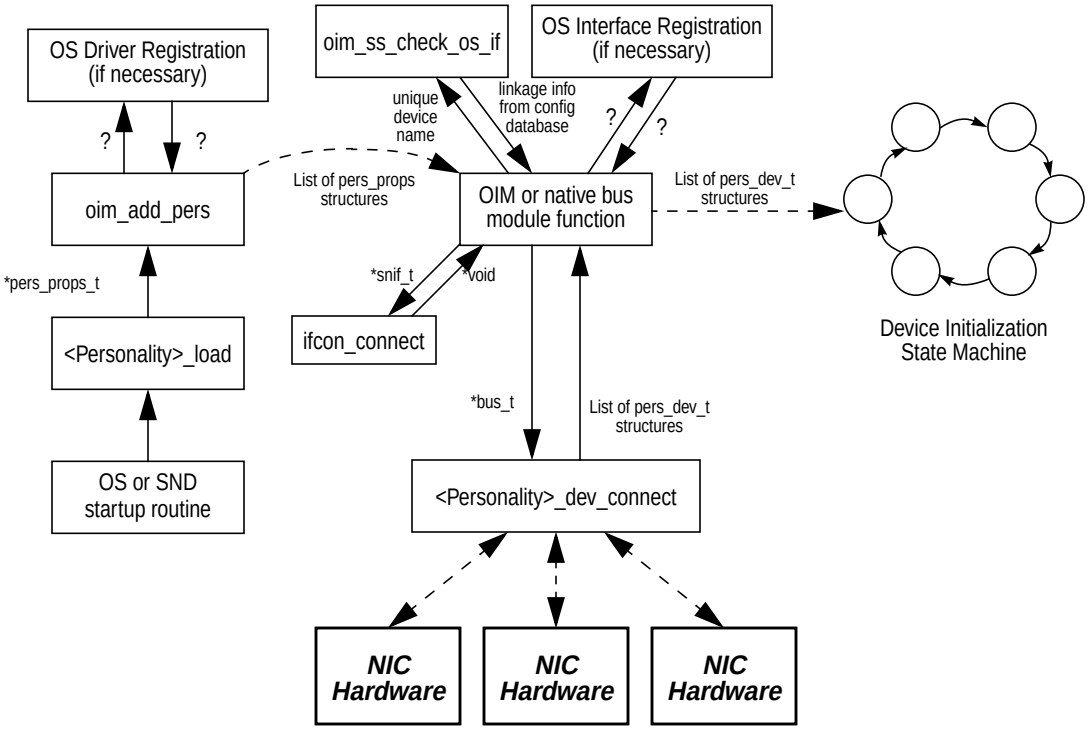


Figure 1.4 Sequence of events for registering a personality module and binding it to a hardware adapter

as the device ID, and they will then probe for their devices. If the personality module recognizes the device (PCI and EISA) or finds any of its devices (VME) it will return a pers_dev_t structure or linked list of pers_dev_t structures. For each pers_dev_t structure received from the personality module, OIM should check to see if there should be a corresponding interface registered with the OS. If so, the interface should passed to the NIF or SIF (as appropriate) to be registered with the OS. If the interface is not to be registered with the OS, then it is either a child device of a PVD or a PSD, and its initialization will be completed when the PVD's initialization is completed. The initial sequence of events for registering a network personality module and binding it to a hardware instance is shown in figure 1.4. The remainder of the initialization proceeds differently depending on the type of the personality module.

1.4.1.1 PPD's with associated OS interfaces

S/NIF should initialize an ifcon_info_t structure, then call ifcon_connect() to get an ifcon handle. S/NIF can then call ifcon_send_enable() to transition the device through the Stack_Connected state to the Enabled state. Alternatively, ifcon_send_stack_connect() can be used to transition the device to the Stack_Connected state, and ifcon_send_enable() called later to complete initialization. The device initialization state machine is shown in figure 1.5.

From the PPD's perspective, a "stack connect" message is received by its message handler, containing the information it needs to communicate with S/NIF (we know S/NIF is the parent because the PPD has an associated OS interface). The PPD allocates most of the resources it needs to operate, registers its interrupt handler with its parent bus, and responses with a "stack connect response" message which contains the information S/NIF needs to communicate with the PPD. To complete the initialization, the PPD will receive an "enable" message, which in most cases has no associated data. It initializes its hardware, and responds with an "enable complete" message.

1.4.1.2 PPD's without associated OS interfaces

The parent PVD sends a "stack connect" message to the PPD's message handler, containing the information it needs to communicate with the PVD. The PPD allocates most of the resources it needs to operate, registers its interrupt handler with its parent bus, and responses with a "stack connect response" message which contains the information S/NIF needs to communicate with the PPD. To complete the initialization, the PPD will receive an "enable" message, which in most cases has no associated data. It initializes its hardware, and responds with an "enable complete" message.

Certain PPD's may not be intended to ever have OS interfaces. In other words, they may be intended to always be used with one or more PVD's, which in turn might have OS interfaces. The interface to such a PPD is unspecified, so it can be anything the PPD and PVD's agree on.

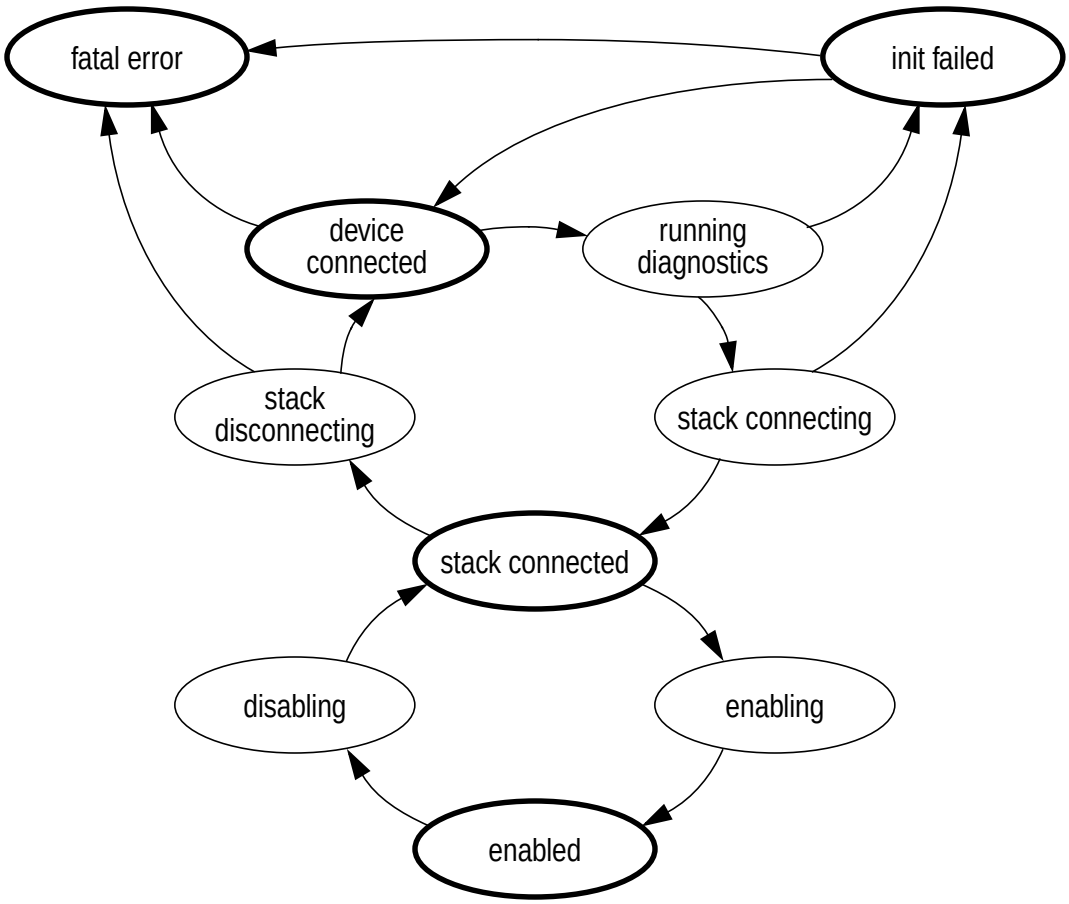


Figure 1.5 Device Initialization State Machine

1.4.1.3 PVD's with associated OS interfaces

S/NIF should initialize an ifcon_info_t structure, then call ifcon_connect() to get an ifcon handle. S/NIF can then call ifcon_send_enable() to transition the device through the Stack_Connected state to the Enabled state. Alternatively, ifcon_send_stack_connect() can be used to transition the device to the Stack_Connected state, and ifcon_send_enable() called later to complete initialization.

From the PPD's perspective, a "stack connect" message is received by its message handler, containing the information it needs to communicate with S/NIF. The PPD allocates most of the resources it needs to operate, and sends a "stack connect" message to its child device(s) which it discovered during the dev_connect phase. When it receives a "stack connect response", it sends its own "stack connect response" to the S/NIF. To complete the initialization, the PPD will receive an "enable" message, which in most cases has no associated data. It performs any necessary final initialization, and sends an "enable" message to its child device(s). When an "enable complete" message is received from the child, it sends an "enable complete" message to the S/NIF.

1.4.1.4 PVD's without associated OS interfaces

The parent PVD sends a "stack connect" message to the PVD's message handler, containing the information it needs to communicate with the parent. The PVD allocates most of the resources it needs to operate, and sends a "stack connect" message to its child device(s) which it discovered during the dev_connect phase. When it receives a "stack connect response", it sends its own "stack connect response" to the parent PVD. To complete the initialization, the PVD will receive an "enable" message, which in most cases has no associated data. It performs any necessary final initialization, and sends an "enable" message to its child device(s). When an "enable complete" message is received from the child, it sends an "enable complete" message to the parent PVD.

1.4.1.5 PSD's

A PSD registers a single message handler during its load entry point, with its service name as the handler name. PVD's or PPD's that wish to use the PSD send a "bind" request to the PSD's message handler, and the PSD responds with a message containing a handle for the service along with any other necessary information. The PSD may also register a separate message handler for each PVD or PPD which binds to it.

The interface between a PSD and PPD/PVD's is not strictly defined, except in the general terms above. It is recommended that the interface to a PSD be restricted to messaging only. Future SND specifications may allow a PSD to reside in user space, and communication with the kernel portion of SND via messages.

1.4.2 Interrupt Handling

To register an interrupt handler, a Personality Physical Driver (PPD) builds a BUS_ISR_BIND message, and sends it to its parent bus. Only a PPD can register an ISR (a natural consequence of the fact that only PPD's have a handle to a bus module). Once an acknowledgment has been received from the parent bus, the PPD can enable interrupts from the hardware. The parent bus registers with its parent bus if it is a bus bridge module, or registers an interrupt handler with the OS if it is a native bus module.

When an interrupt occurs, the OS calls the native bus module's interrupt handler, which is responsible for performing any OS specific interrupt functions. As part of the BUS_ISR_BIND message, the PPD provides a set of functions to facilitate this processing, such as checking if the hardware generated the interrupt (used for shared interrupt support), and disabling and enabling interrupts at the hardware level (required by some operating systems). At the appropriate point during this OS specific processing, the interrupt handler for the child bus bridge or PPD will be called. If the child is a bus bridge, it will perform its own interrupt process-

ing and in turn call its child interrupt handler. Ultimately the PPD's interrupt handler will be called, and it will handle and clear any pending interrupts at the adapter.

1.4.3 Packet Transmission

The OS transmits a packet by calling into the NIF in an OS specific manner, typically through a function pointer the NIF has previously provided to the OS. After acquiring the OS structure describing the packet, the NIF should check its local state for the relevant interface to see if packet transmission can even be attempted (i.e. make sure the hardware is enabled, and the link is up). If the packet can't be transmitted due to the interface state, it should be dropped. Otherwise, if required by the OS, the NIF should allocate memory for the network header and build the network header. In addition, memory should be allocated for any headers required by lower level PVD's or PPD's (previously determined from LAN_STACK_CON_RESP). This allocation can be done either in the NIF or in the NBDM -- generally if the NIF must allocate space for a network header, it should allocate memory for the other headers at the same time. The responsibility for header allocation can be left unspecified because both the NIF and the NBDM are in the Zone region, so they can be implemented with the appropriate knowledge of each other.

Next, bdm_net_pkt_prepare() should be called to map the OS packet representation into SND's internal representation (packet_t). If the BDM is responsible for allocating the headers for the lower level personalities, it will be done in bdm_net_pkt_prepare. Finally, the packet_t is passed to the send_pkt() entry point for the child PPD, previously determined from the LAN_STACK_CON_RESP. If there is no space available to queue the packet, the PPD send_pkt() entry point should return SND_Pending. If there is something wrong with the hardware (hardware is not initialized, or the link is down) the PPD should return SND_Fail. Otherwise, the PPD calls bdm_tx_buf_sync() which performs the final cache synchronization, then calls a PPD provided callback routine which will attempt to queue the packet to the hardware and return SND_Success. If the PPD returns SND_Pending, then packet can be dropped or placed on a wait queue. SND_Fail indicates the packet should be dropped. Figure 1.6 shows an example of the data flow for a transmitted packet.

The above explanation makes the assumption that there are no PVD's in the data path. The send_pkt() entry point for a PVD performs the same functions as the PPD send_pkt() entry point, except that it will probably need to call a lower level PVD or PPD, and it is not allowed to call bdm_xmit_setup().

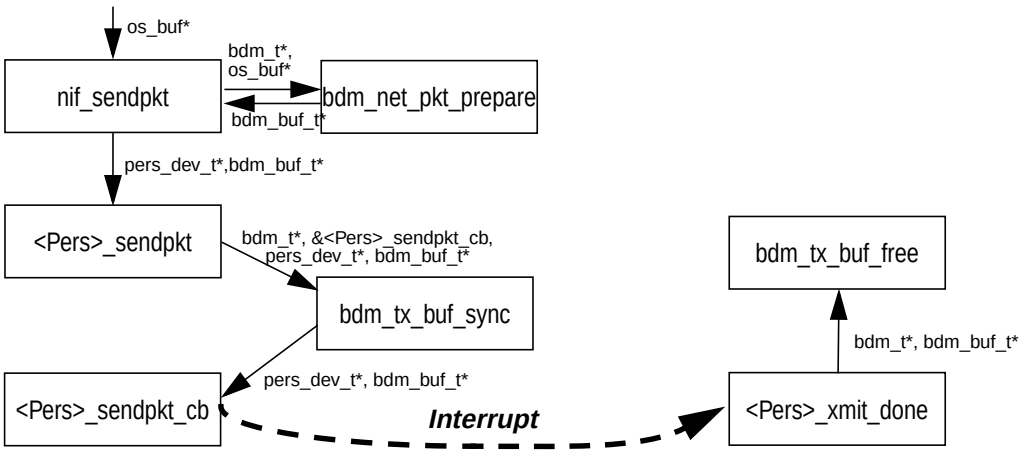


Figure 1.6 Example data flow for transmitted packet

1.4.4 Packet Reception

Typically, a received packet originates from a PPD, and the received process is started during the PPD's interrupt handler by checking the receive pending queue (or equivalent). If the adapter has indicated any sort of error receiving the packet, it should be discarded unless that instance of the PPD is in promiscuous mode. Assuming the packet was received successfully, bdm_rcv_buf_sync() should be called, which will synchronize (purge) the cache and call a PPD provided callback function. If there is a hardware specific header associated with the packet (in addition to the network header) the callback should strip it by calling bdm_pkt_consume_hdr(). Then the recv_pkt() entry point provided by the last LAN_STACK_CON_MSG should be called.

In this case, assume the entry point is the NIF's receive entry point (in other words, there are no PVD's between the PPD and the NIF). The NIF will call bdm_rcv_buf_giveaway, which copies or unmaps the receive buffer as appropriate, and returns an OS buffer containing the data from the received packet. Any remaining OS specific processing should be done at this time, including protocol demultiplexing, and then the OS buffer should be passed to the OS in the appropriate manner. Figure 1.7 shows an example of data flow for a typical received packet. Note that the function names for the nif and personality are examples, and actual names may differ.

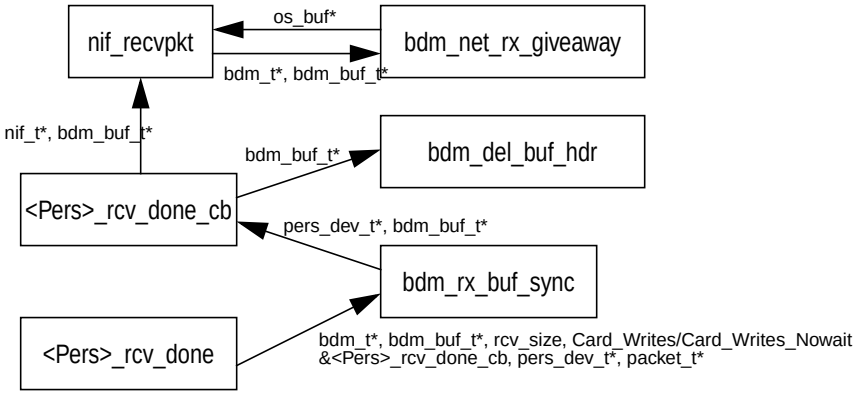


Figure 1.7 Example data flow for received packet

1.4.5 SCSI Commands

The OS requests the SIF to perform a SCSI command in an OS specific manner. Upon receiving the command, the SIF should call bdm_scsi_cmd_prepare() to translate the OS structure associated with the command into a portable representation (bdm_buf_t) that the rest of the SND environment can work with. The SIF should then pass the bdm_buf_t structure to the <Pers>_scsi_cmd() entry point previously registered by the child personality during the stack connect phase. The child personality will call bdm_tx_buf_sync(), which will flush the cache associated with the buffers for the SCSI command. When the personality's callback routine (provided in the call to bdm_tx_buf_sync()), it will execute the SCSI command, including setting the timer if a timeout for the command was specified.

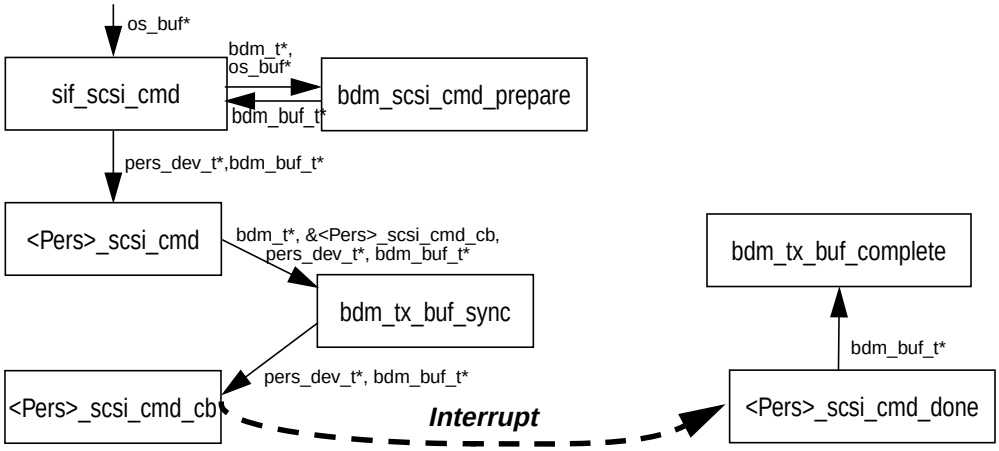


Figure 1.8 Example data flow for SCSI command

When the command is complete, the callback routine contained within the bdm_buf_t structure is called (using the macro bdm_tx_complete()), allowing the BDM and SIF to complete the SCSI command processing.

2.0 Personality Implementation Guide

2.1 Level 1, Level2, Level3 functions

To provide a common structure across personality modules, it is recommend that the functions of a PVD or PPD be partitioned into level 1, level 2 and level 3 functions as follows:

- All external interface functions are considered level 1 functions. There are also some level 1 functions which are used to implement the message handler.
- Level 2 functions provide building blocks to implement level 1 functions.
- Level 3 functions are internal functions of unspecified purpose, used to implement level 2 functions.

2.2 MP Issues

SND 3.0 allows two approaches to MP synchronization within personalities: region based locking and fine grained locking. Region based locking uses a single lock per personality device instance, which is acquired every time a function in the personality module is called, and released every time that function returns. In most cases, this locking and unlocking is performed transparently to the personality module by the Zone or Ifcon, but sometimes the personality must acquire the lock itself. All personalities must support region based locking.

Additionally, a personality can choose to implement fine grained locking which can supersede the region based locking approach. The Zone the personality runs within then determines whether the region based locking scheme or the fine grained locking scheme will be used.

2.2.1 Region Based Locking

Personalities use a set of region locking services to support the region based locking. Table x describes these services briefly, and they are fully documented in the Core reference section of this document.

Service Name	Description
snd_create_pregion_lock	Creates a region lock for a given personality instance.
snd_free_pregion_lock	Frees the region lock of a given personality instance.
SND_LOCK_PREGION_C	Acquire the region lock of a given personality instance, given a pointer to its pers_dev_t structure.
SND_UNLOCK_P_REGION_C	Release the lock acquired with SND_LOCK_PREGION_C.
SND_LOCK_PREGION_B	Acquire the region lock of a given personality instance, given a pointer to its system bus_t structure.
SND_UNLOCK_PREGION_B	Release the lock acquired with SND_LOCK_PREGION_B.

TABLE 1. SND Region Locking Services Summary

The <pers>_dev_connect entry point is responsible for creating the region lock for each device instance, even if it wishes to use fine grained locking. If the personality uses the snd_init_card_struct() service to allocate and initalize the device specific structure for each device instance, then the region lock will be created automatically. The <pers>_dev_connect frees the region lock, either explicitly or by calling snd_free_card_struct().

When the personality registers its message handler, it can specify its region lock as the lock to be used when its message handler is called (pC->pers_lock). Personalities must always explicitly release the region lock (if

held) before sending messages to other modules. If a personality uses any timer services (i_timer), its time-out function is responsible for acquiring and releasing the region lock when called. All external entry points to the personality must be called while holding the region lock for the relevant personality device instance, unless the personality is a PSD, in which case it must perform all its locking itself. A PSD still must support region based locking, even if it supports fine grained locking since the OS may not support fine grained locking

2.2.2 Fine Grained Locking

Personalities which wish to support fine grained locking must be written so that they apparently make all the necessary calls for both region based locking and fine grained locking. The Zone will then cause one or the other set of locking calls to have no effect, selecting the appropriate locking method for the personality (see the Zone implementation guide). A Zone might choose not to support fine grained locking either for decreased development time, or because the locks implemented by the OS it supports are poorly suited to fine grained locking (Solaris). Table x shows the locking services a personality should use to implement fine grained locks.. Within a personality region, i_lock() and i_unlock() are undefined.

Service Name	Description
i_create_lock	Create a spinlock
i_free_lock	De-allocate a spinlock
i_plock	Acquire a spinlock allocated with i_create_lock() from within a Personality module.
i_punlock	Release a spinlock acquired with i_plock().

TABLE 2. Personality Fine Grained Locking Services

2.3 TX Interrupts

There are two basic approaches to detecting transmit completions: interrupts and polling. Polling has the advantage of having relatively low CPU impact, whereas interrupts provide better latency with significant CPU overhead. All PPD’s must be able to support interrupt driven transmit completion, and it is recommended that PPD’s be capable of selecting between polling and interrupt driven transmit completion based on the requirements of the OS and Zone.

The Zone communicates its requirements using the tx_buf_policy field in the bdm_t structure. tx_buf_policy can have the following values:

Immediate_Return Buffers must be returned as quickly as possible. Turn on transmit completion interrupts.

Delay_Return Buffers need not be returned as soon as possible, but should still be returned at the next “convenient” opportunity. Usually this means checking for TX completions the next time a TX is requested.

Select_Return Some buffers must be returned quickly, but others can be delayed. The IMM_RETURN flag will be set in a buffer which must be returned immediately. PPD’s should use an appropriate strategy to ensure the buffer requirements are met. The simplest approach is to turn on transmit interrupts. The most efficient approach may be to turn interrupts on and off depending on whether a IMM_RETURN buffer is currently posted. Some hardware supports requesting interrupts for selected operations, which would be ideal for this mode.

2.4 TX Flow Control

Because of the varying requirements of operating systems and the varying capabilities of hardware, some method of flow control must exist between the S/NIF and the personalities. SND flow control is implemented through a combination of the return value from the <Pers>_send_pkt and <Pers>_scsi_cmd entry points, and calls by the personality to its parent’s tx_stop and tx_restart entry points (provided during the stack connect phase).

2.4.1 Return Value

The personality should return SND_Fail from <Pers>_send_pkt or <Pers>_scsi_cmd if there are not enough resources to handle the request. NIF personality modules are not expected to provide any internal software queuing capability, so if the hardware queue does not have enough space for the request, SND_Fail should be returned. SIF personalities are expected to provided significant internal software queuing, and so a SIF personality should only return SND_Fail if its internal software queue is full. Upon receiving SND_Fail from a request, the S/NIF will assume the device is “flow blocked” and will not send any further requests to that device until the S/NIF’s tx_restart entry point is called.

2.4.2 tx_stop

If the personality determines that there are not enough resources to satisfy a request (transmit a packet or SCSI command), but it is already in the callback from bdm_tx_buf_sync(), is should call the tx_stop() entry point provided during the stack connect phase. This tells the parent module (and ultimately the S/NIF) that the device is flow blocked, as well as providing the request which could not be satisfied back to the S/NIF where it can be retried. The S/NIF will not attempt to send any further requests to that device until the S/NIF’s tx_restart entry point is called.

2.4.3 tx_start

Once the personality has determined that its device is no longer flow blocked (usually after taking a transmit completion interrupt) it must call the tx_start() entry point provided during the stack connect phase. This

will cause the S/NIF to send any commands or packets which were queued up waiting for the device, then resume normal operation

2.4.4 Interrupt Considerations for Flow Control

If a personality is not using TX interrupts, and is instead polling for TX completions on every send request, when the device enters the flow blocked state, the S/NIF will no longer be calling the send entry point. This effectively means the personality will never again check for completions, which is a problem since only be checking for completions will the personality know when to call the tx_start() entry point.

The solution is to enable TX interrupts upon entering the flow blocked state, then disable them when the device is no longer flow blocked (assuming they weren't enabled in the first place). State will be required to satisfy the interrupt requirements of the BDM and the flow control simultaneously. For example, if the BDM specified Immediate_Return for its buffers, and the device goes into a flow blocked state, then the personality disables interrupts when it is no longer flow blocked, the BDM's requirements will no longer be satisfied.

2.5 Level 1 Interface Functions:

Function Name	Synopsis
<Pers>_dev_connect	Recognize device, create and initialize corresponding device specific structure(s).
*<Pers>_dev_disconnect	Release all remaining resources associated with a particular logical device, and then free the control structure related to that device.
<Pers>_stack_connect	Allocate necessary resources, register interrupt handler
<Pers>_stack_disconnect	Release all resources, deregister interrupt handler
<Pers>_enable	Bring device to Enabled state (HW and SW fully operational)
<Pers>_disable	Bring device to Disabled (Stack_Connected) state.
<Pers>_send_pkt	Send a network packet, prepared by the NIF.
<Pers>_scsi_cmd	Perform the requested SCSI operation, prepared by the SIF.
*<Pers>_isr	Process an interrupt generated by a device.
*<Pers>_check_if_int_pending	Check if a device has generated an interrupt.
*<Pers>_disable_ints	Disable interrupts from a device.
*<Pers>_enable_ints	Re-enable interrupts from a device.

TABLE 3. Personality Level 1 Functions

* Sufficient documentation is in the Personality Interface Reference

Function:

<Pers>_dev_connect

Prototype:

```
pers_dev_t *<Pers>_dev_connect(bus_t *pbus, snd_dev_id_t *dev_id);
```

Synopsis

Recognize device, create and initialize corresponding pers_dev_t structure(s).

Parameters

pbus	Pointer to a bus_t structure created by the parent bus of the offered device
dev_id	ID structure which contains the appropriate information to identify the offered device on the parent bus. See the description of snd_dev_id_t for more details.

Return Value

Pointer to a pers_dev_t structure or list of pers_dev_t structure (one per device created). Returning NULL indicates either the device was not recognized, no devices were found, the device could not be initialized.

Detailed Description

Personality Physical Drivers

On buses which can be enumerated, such as PCI, EISA, and SBus, a bus module will call <Pers>_dev_connect with the *dev_id* for each device encountered on that bus. The <Pers>_dev_connect function is responsible for determining whether it recognizes the device (meaning it can drive the device). If it recognizes the device, it must perform the following tasks:

- Create a device specific structure for the device, which includes a pers_dev_t structure.
- Call snd_init_card_struct to initialize portions of the card structure, create the personality region lock, and create a memory pool.
- Build an SND open firmware ID string for the device by calling snd_build_OF_path
- Register a message handler for the device

Finally, if all of the above steps were completed successfully, the pointer to the allocated pers_dev_t structure should be returned. If the device was not recognized, NULL should be returned. If an error occured initializing the device, NULL should be returned.

Buses which cannot be enumerated, such as VME and ISA, will not be passed a *dev_id* value. For each personality associated with the bus, that personalities <Pers>_dev_connect function will be called exactly once. The <Pers>_dev_connect should probe for supported devices, and for each device it discovers, it should perform the same tasks listed above for enumerable buses.

The resulting pers_dev_t structures should be connected as a linked list, and a pointer to the head of the list should be returned. If no devices were found, or no devices could be initialized, NULL should be returned.

Personalities for multichannel or multifunction boards should create one pers_dev_t structure per logical device on the adapter, and return the pers_dev_t structures as a linked list.

Personality Virtual Drivers

The <Pers>_dev_connect function for a PVD will be called with snd_dev_id_t containing a device name. Unlike a PPD, the PVD is not expected to validate or identify this device. It is the responsibility of the caller to decide which PVD is associated with the device. Upon being called, the PVD should perform the following tasks:

- Create a device specific structure for the device, which includes a pers_dev_t structure.
- Call snd_init_card_struct to initialize portions of the card structure, create a region lock, and create a memory pool. Use NULL as the bus pointer.
- Register a message handler for the device, using the device name as the message handler name.
- Personality Service Drivers

The resulting pers_dev_t structure should be returned. If any of the required tasks could not be performed, NULL should be returned.

Personality Service Drivers

When the <Pers>_dev_connect function for a PSD is called, the PSD should allocate a control structure for itself, allocate a region lock, and register a message handler. Users of the PSD will send bind requests to the PSD’s message handler later which will be handled in a manner specific to the PSD.

Function:

<Pers>_stack_connect

Prototype:

```
void <Pers>_stack_connect(pers_dev_t *pC,
                          <Lan|SCSI>_stack_con_msg_t *stack_con_msg);
```

Synopsis

Allocate necessary resources, register interrupt handler.

Parameters

pC	Pointer to the pers_dev_t structure associated with the desired interface.
stack_con_msg	Pointer to the lan_stack_con_msg_t or scsi_stack_con_msg_t message data.

Return Value

None

Detailed Description

<Pers>_stack_connect() is used to handle LAN_STACK_CON_MSG and SCSI_STACK_CON_MSG message types. When either of these message types is received, the personality must first save off the source handle that the message was sent with. *This is not the same as the parent_handle contained within the message!* The source handle and parent handle will sometimes be different. After saving the source handle, the personality can call <Pers>_stack_connect to finish handling the message.

In <Pers>_stack_connect, the personality must:

- Allocate a stack_con_resp message of the appriate type (can be on the stack)
- Fill in the BDL structure contained within the message
- Use the BDL structure to call bus_dma_init.
- Use the BDL structure to call bdm_connect and get a BDM handle.
- Call <Pers>_allocate_resources
- Get the adapters MAC address (LAN adapters only).
- Register the interrupt handler
- Fill in all required fields of the stack_con_resp message. This can be interspersed with the above operations in whatever manner is convenient.
- Send response to the ***source handle*** the stack_con_msg was received from

The stack connect response does not need to be sent synchronously with the receipt of the stack connect message.

Function:

<Pers>_stack_disconnect

Prototype:

```
void
<Pers>_stack_disconnect(pers_dev_t *pC,
                        <lan|scsi>_stack_discon_msg_t *stack_discon_msg);
```

Synopsis

Release all resources, deregister interrupt handler.

Parameters

pC	Pointer to the pers_dev_t structure associated with the desired interface.
stack_discon_msg	Pointer to the lan_stack_discon_msg_t or scsi_stack_discon_msg_t message data.

Return Value

None

Detailed Description

<Pers>_stack_disconnect() is used to handle LAN_STACK_DISCON_MSG and SCSI_STACK_DISCON_MSG message types. When either of these message types is received, the personality can call <Pers>_stack_disconnect to handle the message.

In <Pers>_stack_disconnect, the personality must:

- De-register the interrupt handler
- Allocate a stack_discon_resp message of the appriate type (can be on the stack)
- Call bdm_disconnect()
- Call bus_dma_done
- <Pers>_deallocate_resources
- Fill in all required fields of the stack_discon_resp message.
- Send response to the ***source handle*** that was saved off during the initial stack connect message

The stack disconnect response does not need to be sent synchronously with the receipt of the stack disconnect message.

Function:

<Pers>_enable

Prototype:

```
void <Pers>_enable(pers_dev_t *pC, <lan|scsi>_enable_msg_t *enable_msg);
```

Synopsis

Bring device to Enabled state (HW and SW fully operational).

Parameters

pC	Pointer to the pers_dev_t structure associated with the desired interface.
enable_msg	Pointer to the lan_enable_msg_t or scsi_enable_msg_t message data.

Return Value

None

Detailed Description

<Pers>_enable() is used to handle LAN_ENABLE_MSG and SCSI_ENABLE_MSG message types. When either of these message types is received, the personality can call <Pers>_enable to handle the message.

In <Pers>_enable, the personality should:

- Call <Pers>_hardware_reset
- Call <Pers>_setup_resources
- Call <Pers>_init_hardware
- Send response to the ***source handle*** that was saved off during the initial stack connect message

The enable response does not need to be sent synchronously with the receipt of the enable message.

Function:

<Pers>_disable

Prototype:

```
void     <Pers>_disable(pers_dev_t     *pC,     <lan|scsi>_disable_msg_t
*disable_msg);
```

Synopsis

Bring device to Disabled (Stack_Connected) state.

Parameters

pC	Pointer to the pers_dev_t structure associated with the desired interface.
disable_msg	Pointer to the lan_disable_msg_t or scsi_disable_msg_t message data.

Return Value

None

Detailed Description

<Pers>_disable() is used to handle LAN_DISABLE_MSG and SCSI_DISABLE_MSG message types. When either of these message types is received, the personality can call <Pers>_disable to handle the message.

In <Pers>_enable, the personality should:

- Call <Pers>_stop_hardware
- Call <Pers>_cleanup_resources
- Send response to the ***source handle*** that was saved off during the initial stack connect message

The disable response does not need to be sent synchronously with the receipt of the disable message.

Function:

<Pers>_send_pkt

Prototype:

```
i_status_t <Pers>_sendpkt(pers_dev_t *pC, bdm_buf_t *buf);
```

Synopsis

Send a network packet, prepared by the NIF.

Parameters

pC Pointer to the pers_dev_t structure associated with the interface on which the network packet should be transmitted

buf Buffer structure containing the packet data to be transmitted, along with any upper layer headers.

Return Value

If the packet was successfully transmitted or queued to the hardware, SND_Success should be returned. If there was an error transmitting or queuing the packet SND_Fail should be returned.

Detailed Description

Personality Physical Drivers

Upon being called, the packet buffer will contain all the packet data, including standardized network headers such as ethernet or FDDI MAC headers. Space will be provide for hardware specific header data if the personality specified an extra_tx_hdr_bytes value during the stack connect phase. The first thing the <Pers>_send_pkt function should do is check to see if there are enough resources to queue the packet to the hardware. Then the PPD must fill in its header bytes (if any were specified). Finally, bdm_tx_buf_sync must be called to synchronize the cache associated with the buffer (and possibly to map the buffer as well). After buffer synchronization is complete, the BDM will call the callback which the personality provided in the call to bdm_tx_buf_sync.

The callback function must program the hardware to transfer the packet. The physical addresses of the buffer fragments are contained within a scatter-gather list in the bdm_buf_t structure. Once bdm_tx_buf_sync has been called, there is no further opportunity to return status to the NIF, so errors must be handled differently. If an error occurs which would require the packet to be resent later (resource shortage) the PPD should call the tx_stop entry point specified in the stack_connect message. If the packet should not be resent, the PPD should call bdm_tx_buf_error() with an appropriate error code.

NIF’s are expected to build valid ethernet and FDDI headers, but are not expected to build Fibre Channel headers. Therefore, Fibre Channel networking personalities will need to build the FC headers themselves.

Personality Virtual Drivers

PVD’s should fill in their relevant headers in the space provided (established during stack connect), and call the <Pers>_send_pkt entry point of their child device.

Function:

<Pers>_scsi_cmd

Prototype:

```
i_status_t <Pers>_scsi_cmd(pers_dev_t *pC, bdm_buf_t *buf);
```

Synopsis

Perform the requested SCSI operation, prepared by the SIF.

Parameters

pC Pointer to the pers_dev_t structure associated with the interface on which the SCSI command should be transmitted

buf Buffer structure containing the SCSI command to be transmitted, along with any necessary data buffers.

Return Value

If the SCSI command was successfully transmitted or queued to the hardware, SND_Success or SND_Pending should be returned. If there was an error transmitted or queuing the command, SND_Fail should be returned.

Detailed Description

TBD

2.6 Level 2 Interface Functions:

Function Name	Synopsis
<Pers>_allocate_resources	Allocate all memory, handles, etc that will be used by the personality while it is in the Enabling, Enabled and Disabling states.
<Pers>_deallocate_resources	Free the resources allocated by <Pers>_allocate_resources().
<Pers>_reset_hardware	Reset the device hardware to a state similar to powerup. Configuration information provided by the BIOS or OS must be retained.
<Pers>_setup_resources	Initialize all structures to “ready to run” state.
<Pers>_cleanup_resources	Clean up any operations which are in process on the device or within the personality. Prepare for a “disable” or “reset”.
<Pers>_init_hardware	Initialize the hardware to a “ready to run” state
<Pers>_stop_hardware	Stop hardware from performing any actions, including DMA and interrupts.

TABLE 4. Personality Level 2 Function

Function:

<Pers>_allocate_resources

Prototype:

```
i_status_t <Pers>_allocate_resources(pers_dev_t *pC);
```

Synopsis

Allocate all memory, handles, etc that will be used by the personality while it is in the Enabling, Enabled and Disabling states.

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

SND_Success if all the resources were allocated, otherwise SND_Fail.

Detailed Description

<Pers>_allocate resources is expected to allocate most of the resources the personality will need in order move to the Enabled state. The following resources should not be allocated in <Pers>_allocate_resources:

- bdm handle
- receive buffers
- call to bus_dma_init

And the following are examples of resources which should be allocated in <Pers>_allocate_resources:

- Memory not intended for TX or RX buffers
- Mapped memory for control structures
- PIO handles

Function:

<Pers>_deallocate_resources

Prototype:

```
void <Pers>_deallocate_resources(pers_dev_t *pC);
```

Synopsis

Free the resources allocated by <Pers>_allocate_resources().

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

None

Detailed Description

Just free whatever was allocated in <Pers>_allocate_resources() in reverse order.

Function:

<Pers>_reset_hardware

Prototype:

```
void <Pers>_reset_hardware(pers_dev_t *pC);
```

Synopsis

Reset the device hardware to a state similar to powerup. Configuration information provided by the BIOS or OS must be retained.

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

None

Detailed Description

As stated above, the objective of this function is to return the hardware to a state as close as possible to powerup. Configuration information, such as PCI memory base register settings, must be preserved and restored in the hardware. Some hardware can be reset without losing this information, but many devices will require the <Pers>_reset_hardware function to read all the configuration data, save it off, and then restore it after the reset is complete.

Upon return from this function, the device should be ready for access. If a device requires “quiet time” after a reset, such time should be incorporated into the <Pers>_reset_hardware function.

Function:

<Pers>_setup_resources

Prototype:

```
i_status_t <Pers>_setup_resources(pers_dev_t *pC);
```

Synopsis

Initialize all structures to “ready to run” state.

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

SND_Success if the resources were initialized properly, otherwise SND_Fail

Detailed Description

<Pers>_setup_resources initializes all structures to a correct and stable state, which may include zeroing areas of memory, and initializing counters and pointers to default values.

Function:

<Pers>_cleanup_resources

Prototype:

```
void <Pers>_cleanup_resources(pers_dev_t *pC);
```

Synopsis

Clean up any operations which are in process on the device or within the personality. Prepare for a “disable” or “reset”.

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

None

Detailed Description

The main intention of the <Pers>_cleanup_resources function is to cleanup up and TX, RX, or SCSI buffers posted to hardware or software queues within the driver. These buffers must be gracefully freed or returned to the OS (as appropriate) so that when the structures are reinitialized with <Pers>_setup_resources, the buffers will not be lost.

Function:

<Pers>_init_hardware

Prototype:

```
i_status_t <Pers>_init_hardware(pers_dev_t *pC);
```

Synopsis

Initialize the hardware to a “ready to run” state

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

SND_Success if the hardware was initialized properly, otherwise SND_Fail.

Detailed Description

As the name implies, this function completes initialization of the device. In addition to programming appropriate registers, and issuing appropriate commands, it is at this time that receive buffers should be posted to network adapters.

Function:

<Pers>_stop_hardware

Prototype:

```
void <Pers>_stop_hardware(pers_dev_t *pC);
```

Synopsis

Stop hardware from performing any actions, including DMA and interrupts.

Parameters

pC Pointer to the pers_dev_t structure associated with the desired interface.

Return Value

None

Detailed Description

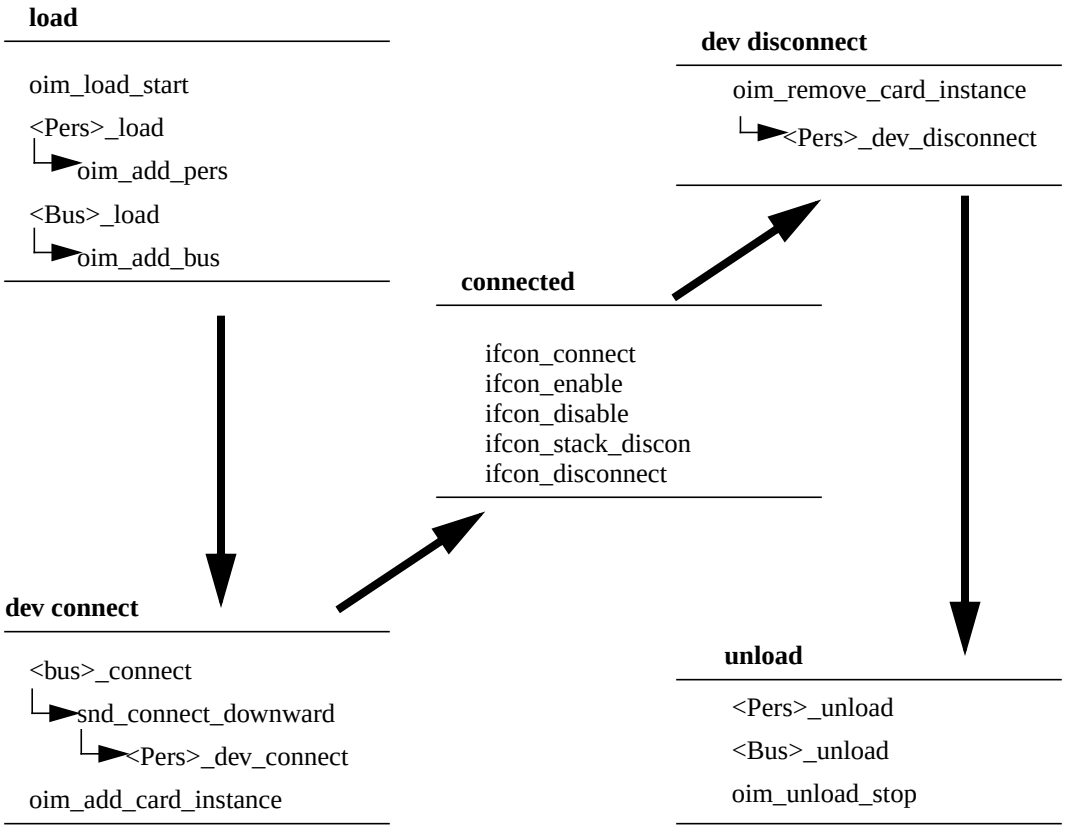
<Pers>_stop_hardware should gracefully disable the hardware, finishing up by calling <Pers>_reset_hardware to leave the device in a “close-to_powerup” state. Before calling <Pers>_reset_hardware, the transmit side of the adapter should be gracefully stopped, followed by the receive side.

The intention of the graceful shutdown is to avoid system problems when adapters are reset in the middle of a transaction.

3.0 Zone Implementation Guide

3.1 SND Phases of Operation

There are five general phases of operation for an SND environment: load, dev connect, connected, dev disconnected, and unload. Figure 7.1 shows the types of operations which should occur in each phase.



Exceptions:

- OIM can implement oim_add_pers() and oim_add_bus() so that they call oim_load_start() if it has not already been called.
- Some operating systems may call directly into SND native bus modules, in which case the call to <bus>_connect is not necessary. snd_connect_downward() will still be called.
- Some operating systems will require a device to be initialized immediately after it is discovered during the dev_connect phase. If this is the case, then the boundary between dev_connect and connected can be ignored, and the device can be initialized using the ifcon functions before the next device is dev connected. NT may required this.

Figure 3.1 SND Phases of operation

SND Zones should conform to the specified phases in order provide the expected environment to personality modules, in addition to providing a similar structure to other Zones.

3.2 Suggested Steps to Implement Zone

- Implement OSM functionality
- Design and implement native bus module for target bus.
- Design and implement OIM device recognition code
- Design and implement OIM stable storage facility
- Design and implement SIF
- Design and implement NIF
- Design and implement BDM

3.3 Relationship of BUS, OIM, NIF and SIF

During the dev connect phase, the main function of the BUS is to probe for and recognize devices or, if the OS probes itself, to recognize devices the OS discovered. Device recognition is done using snd_connect_downward(), which will "offer" the device to all installed bus bridge and personality modules with the appropriate parent bus type. It will return a list of one or more pers_dev_t structures which then need to be distributed between the OIM, NIF and SIF.

The OIM needs a record of all devices. For each pers_dev_t structure returned by snd_connect_downward, the BUS should call oim_add_card_instance(), which should create a control structure (probably called oim_t) which points to the pers_dev_t structure, and add the structure to a list of oim_t structures maintained by the OIM. Additionally, if the device associated with the pers_dev_t structure is a networking device, oim_add_card_instance() should call nif_add_card_instance() which will return a pointer that oim_add_card_instance() should save in the oim_t structure for later use if a call to nif_remove_card_instance() is necessary. If the device associated with the pers_dev_t structure is a storage device, sif_add_card_instance() should be called, which will perform equivelant functions to nif_add_card_instance().

oim_add_card_instance() can check the card_mod_props.dev_type field in the pers_dev_t structure to know whether it should call nif_add_card_instance() or sif_add_card_instance(). If the dev_type field is SCSI_Dev, then obviously sif_add_card_instance() should be called. If the dev_type field is LAN_Dev, or IP_Dev, then nif_add_card_instance() should be called. Finally, if the dev_type field is FC_Dev or ATM_Dev (looking towards the future), neither should be called, on the assumption that an upper level PVD will be running that device. Figure 7.2 shows a data flow diagram illustrating these relationships..

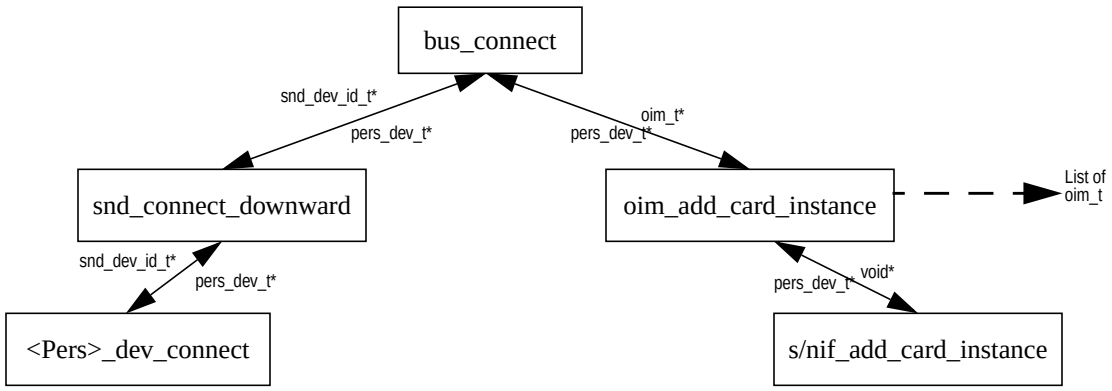


Figure 3.2 Data Flow Diagram for Device Distribution Among BUS, OIM, NIF, and SIF

When implementing oim_add_card_instance, sif_add_card_instance, and nif_add_card_instance, bear in mind that the moment the device is discovered might not be the best time to register the device with the actual OS storage or network subsystem. Sometimes operating systems probe for devices before all the kernel subsystems are initialized. For example, an HP/UX implementation would call nif_add_card_instance from its attach routine, but would not perform the “if_attach” call until later, during the “nif_init” routine.

3.4 Determining whether a device is a PPD, PVD or PSD

All devices which are not PSD’s or PVD’s are PPD’s. Therefore the method for identifying a PPD is to eliminate the possibity of it being a PVD or PSD. A PSD will have a dev_type of Service_Dev in its card_mod_props structure. If the device is not a PSD, then check the bus_type in its card_mod_props structure. If the bus_type is NO_BUS_SPEC, then it is a PVD. Otherwise, the device is a PPD.

3.5 Order of initialization for PPD, PVD, PSD

At some point, every registered personality must have its dev_connect entry point called. Due to the fact that PVD’s or PPD’s may depend on the existance of services provided by PSD’s, the OIM must somehow ensure that the dev_connect entry points for all PSD’s are called before any PPD or PVD dev_connect entry points are called. Since no devices are associated with PSD’s, the dev_connect entry point should be called with a device id of NULL.

The OIM can call snd_get_pvd_instances() to probe for PVD device instances. Alternatively, if the OS specifies devices by name, one at a time, the <Pers>_dev_connect function can be called with the device name in the snd_dev_id_t structure, and the personality will create a structure to match. Note that this behavior is different from PPD’s, in that the PVD will not try to recognize the device name. It will blindly create a pers_dev_t structure for it. OIM’s using this alternate approach will probably not support snd_get_pvd_instances, since they would probably not bother to maintain a separate list of PVD device instances.

3.6 Open Firmware Naming and Stable Storage

An important requirement to make SND 3.0 work is some form of stable storage. A PVD or PPD will call the OIM during the dev_connect phase to determine whether the device it controls has an associated OS interface. Later, during the “connected” phase (specifically after receiving a stack connect message), a PVD will call the OIM to determine its child device name.

The OIM will use the PPD or PVD SND message handler name to look up whether the device should have an associated OS interface, and a PVD’s child device will be specified by its message handler name. PVD’s and PPD’s register their open firmware name as their message handler name. Consequently this name must remain constant across multiple initialization cycles, whether due to machine reboots or unloading and reloading the driver. This should not be difficult for PVD’s (they’re virtual anyway), but getting the appropriate behavior with PPD’s will probably involve using hardware information like bus and slot number, or base I/O address.

3.7 Using the Ifcon

The ifcon provides several useful facilities for managing devices, including a state machine for initializing and shuting down devices, and multicast address tracking. Only devices with associated OS interfaces are controlled by the ifcon. For example, the Zone would never use Ifcon to control a PSD, nor would it use the ifcon to control a PPD which was a child device to a PVD (the PVD would then have an associated OS interface, and the PPD would not).

Before using the ifcon services, the Zone must call ifcon_connect() with appropriate parameter to get an ifcon handle. One of the required parameters is a pointer to an ifcon_info_t structure with all possible fields filled out. The structure definition (Core/ifcon.h) should specify which fields must be filled out before the call to ifcon_connect. Remaining fields will be filled in by the ifcon state machine.

SND 3.0 is designed so that it is not necessary to use the ifcon. This will, however, create a huge burden on the Zone developer, and also eliminates the benefits of using the shared code encompassed by the ifcon. Unless there is no way to provide the desired operation for a particular Zone, all Zones should use the ifcon to manage devices. It believed that the ifcon should be adequate for all environments currently targeted for SND 3.0.

3.8 MP Synchronization

The Zone must provide multiprocessor synchronization for all of its modules using SND locking services. In addition, the Zone and Ifcon must provide MP synchronization for PVD's and PPD's, through either the region based locking model, or a fine grained locking model if the personality supports it. If the Zone makes a call to the Ifcon which will result in a call to a PVD or PPD, the Ifcon will handle the region locking. If the Zone calls the PVD or PPD directly (by calling the "send_pkt" or "scsi_cmd" entry points) it must handle the region locking. When a PVD or PPD calls directly into the Zone (for example, with "recv_pkt"), the Zone must release the region lock while it performs its work, and reacquire the lock before returning to the PVD or PPD.

Personalities can be written to support fine grained locking (see the Personality Implementation Guide), but it is up to the Zone whether to allow the fine grained locking to be used. If the Zone whiches to allow fine grained locking, the SND environment must be compiled with -DALLOW_FINE_GRAINED_LOCKING, which will cause the region locking services to check if the CARD_MP_SAFE flag is set, and if so, do nothing. Additionally, this causes i_plock and i_punlock to actually acquire and release locks, instead of doing nothing.

3.9 Implementing the BDM

Maximizing performance should be the primary design consideration when implementing the BDM. This being the case, every OS requires a slightly different approach, due to different strengths and weaknesses of the OS. The biggest BDM issues can all, one way or another, be considered mapping issues.

3.9.1 map vs. copy

There are two basic approaches to getting physical addresses to data. One way is to map data (using some sort of OS service) in its existing buffer, and the other is to copy the data into a buffer which is already mapped. Each approach has its advantages.

Mapping is the best approach for large portions of data. How large is large depends on how much overhead is associated with the system call which creates the mapping. In other words, if the data could be copied into a pre-mapped buffer in less time that it would take to call the system mapping function, then obviously the data should be copied. It is recommended that the BDM be implemented with a map threshold where data buffers larger than the threshold are mapped, and data buffers less that the threshold are copied into pre-mapped buffers (which should be the size of the threshold). The value of the threshold is system dependent and probably will need to be determined empirically, but will almost always be greater than 64 bytes and less than 4096 bytes.

3.9.2 How to map

Since the BDM is part of the Zone, it has great flexibility in deciding what services to use for mapping. The BDM is permitted to call OS mapping functions directly, using information contained within bus structures, except when the device the buffer is intended for is connected to a bus bridge. Bus bridges (such as the 6200 VME to PCI bridge) may need to perform device specific operations to set up the mapping, so if the device is attached to a bus bridge, the bus_dma_map() service must be used to map the buffer (or bus_mm_alloc can be used to preallocate a buffer). The BDM can also use bus_dma_map to map the buffer for devices attached to native buses if the native bus modules implement the <bus>_dma_map entry point (native bus modules are not required to do this). This is a general Zone implementation decision -- the BDM can be written with the knowledge of what decisions were made in the native bus modules.

3.9.3 When to map

Inbound DMA buffers are always mapped at allocation time (before returning from bdm_rx_buf_alloc), but the mapping for outbound DMA buffers is more flexible. Network transmits (outbound DMA operations) can mapped either during bdm_net_pkt_prepare() or bdm_tx_buf_sync(). Mapping during bdm_net_pkt_prepare() is recommended for network packets, since this occurs outside of the Personality region and should result in improved multiprocessor performance. SCSI commands (which have an outbound DMA component, regardless of whether they are read or write) should be mapped during the call to bdm_tx_buf_sync(), since they are likely to be placed on a wait queue after the call to bdm_scsi_cmd_prepare(). The BDM should avoid tying up large quantities of system resources for long periods of time.

3.9.4 SCSI commands in the BDM

A SCSI command always has an outbound DMA components, and sometimes has an inbound DMA component as well (SCSI reads). This does not fit into the existing network oriented paradigm of the BDM. The recommended approach is to use bdm_scsi_cmd_prepare() to initialize the relevant structures and virtual scatter gather lists. Then, during the call to bdm_tx_buf_sync, map all the relevant buffers (inbound and outbound) and perform the outbound cache synchronization. Some systems also require inbound cache synchronization before the DMA takes place. After the command is complete, the callback function contained within the bdm_buf_t should be called, which will free the command in the case of a SCSI write. For a SCSI read, the callback will perform the inbound cache synchronization, free the command, and provide the data to the OS in the appropriate manner.

3.9.5 Handling Receive Network Packets

When an IP or LAN personality receives a network packet, it will generally call bdm_rx_buf_sync() which must perform cache synchronization on the packet. In order to support bus bridges, bus_cache_sync() must be used to sync the buffer, but by looking at the parent bus and system bus of a PPD instance, it is possible for the BDM to intelligently determine whether to call bus_cache_sync() or to directly sync the buffer using OS services. It will invariably be faster to perform the cache sync directly.

After the call to bdm_rx_buf_sync, the personality will strip any hardware specific headers by calling bdm_del_buf_hdr() then call the receive_pkt function which its parent driver registered with it during the stack connect phase. This will frequently be the nif's receive_pkt function, which should (among other things) call bdm_net_rx_giveaway() to translate the bdm_buf_t into an OS specific representation.

The specific functions performed by bdm_net_rx_giveaway depend on what was done in bdm_rx_buf_alloc() when the buffer was allocated. If the bdm_rx_buf_alloc() provided an OS network buffer (such as an mbuf) directly mapped and encapsulated within the bdm_buf_t, then

bdm_net_rx_giveaway() needs to release the mapped, and return the pointer to the original OS buffer. If bdm_rx_buf_alloc() provided a bdm_buf_t from an internally controlled buffer pool, then bdm_net_rx_giveaway() should allocate an OS network buffer and copy the contents of the bdm_buf_t into it.

3.10 Individual Function Implementations

This section will contain suggestions and guidelines for implementing functions within the Zone interface. As issues or uncertainties regarding specific function arise, clarifications will be added here.

3.10.1 OIM, NIF, SIF

3.10.1.1 oim_load_start

3.10.1.2 oim_stop_unload

3.10.1.3 oim_add_pers

3.10.1.4 oim_remove_pers

3.10.1.5 oim_add_bus

3.10.1.6 oim_remove_bus

3.10.1.7 oim_add_card_instance

3.10.1.8 oim_remove_card_instance

3.10.1.9 oim_ss_pvd_probe

3.10.1.10 oim_ss_get_pvd_child

3.10.1.11 oim_ss_check_os_if

3.10.1.12 nif_add_card_instance

3.10.1.13 nif_remove_card_instance

3.10.1.14 sif_add_card_instance

3.10.1.15 sif_remove_card_instance

3.10.2 BDM

3.10.2.1 bdm_connect

3.10.2.2 bdm_disconnect

3.10.2.3 bdm_tx_buf_alloc

3.10.2.4 bdm_tx_buf_sync

3.10.2.5 bdm_tx_buf_complete

3.10.2.6 bdm_tx_buf_free

3.10.2.7 bdm_rx_buf_alloc

3.10.2.8 bdm_rx_buf_peek

3.10.2.9 bdm_rx_buf_sync

3.10.2.10 bdm_rx_buf_free

3.10.2.11 bdm_del_buf_hdr

3.10.2.12 bdm_net_pkt_prepare

3.10.2.13 bdm_scsi_cmd_prepare

3.10.2.14 bdm_net_rx_giveaway

3.10.3 OSM

3.10.3.1 osm_i_malloc

3.10.3.2 osm_i_free

3.10.3.3 osm_i_timeout

3.10.3.4 osm_i_untimeout

3.10.3.5 osm_i_printstr

3.10.3.6 osm_logmsg

3.10.3.7 osm_i_createlock

3.10.3.8 OSM_I_LOCK

3.10.3.9 OSM_I_UNLOCK

3.10.4 BUS

3.10.4.1 <bus>_dma_init

3.10.4.2 <bus>_dma_handle_alloc

3.10.4.3 <bus>_dma_handle_free

3.10.4.4 <bus>_dma_map

3.10.4.5 <bus>_mm_alloc (bus_mm_zalloc will call <bus>_mm_alloc, then zero the memory)

3.10.4.6 <bus>_mm_free

3.10.4.7 <bus>_dma_unmap

3.10.4.8 <bus>_cache_sync

3.10.4.9 <bus>_isr_add

3.10.4.10 <bus>_isr_remove

3.10.4.11 <bus>_pio_map

3.10.4.12 <bus>_pio_unmap

3.10.4.13 <bus>_in_ubit8

3.10.4.14 <bus>_in_ubit16

3.10.4.15 <bus>_in_ubit32

3.10.4.16 <bus>_in

3.10.4.17 <bus>_out_ubit8

3.10.4.18 <bus>_out_ubit16

3.10.4.19 <bus>_out_ubit32

3.10.4.20 <bus>_out

3.10.4.21 <bus>_protect_region_from_dma

3.10.4.22 <bus>_allow_dma_to_region

3.10.4.23 <bus>_disconnect

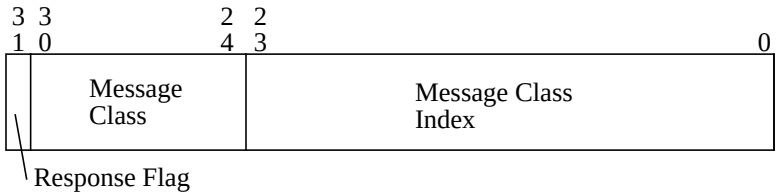
4.0 Bus Bridge Implementation Guide

TBD

5.0 Core Interface Reference

5.1 General Description

5.2 Message Interface



Field name	Bit Positions	Purpose
Message Class Index	0 - 23	Message index within this message class
Message Class	24 - 30	Indicates general class of message
Response Flag	31	Set for message numbers which are intended responses to other messages.

Figure 5.1 Format of an SND Message Number

The messaging subsystem provides a mechanism for modules which have registered a message handler function to exchange messages with other modules which have register message handlers. These messages can be used for any type of information exchange, but the subsystem is implemented with emphasis on simplicity, not performance, so performance critical operations should not use messages. Figure 2.1 shows the format of an SND message number. The maximum size of a normal SND message or response is 32k (128 * 256).

Function Name	Synopsis
i_register_msg_handler	Register a message handler with the message subsystem
i_deregister_msg_handler	Remove a message handler.
i_msg_alloc	Allocate space for message
i_msg_free	Free message space previously allocated with i_msg_alloc.
i_send_msg	Send a message to the message handler associated with the specified handle.
i_send_msg_byname	Send a message to the message handler with the specified name
i_change_handler_name	Change the name of a message handler
i_change_handler_lock	Change the associated lock of a message handler
BLD_MSG_NUM	Macro to create message numbers, based on message class, index within the class, and payload size
BLD_RESP_NUM	Macro to create response numbers for message numbers which have been created with BDL_MSG_NUM.

TABLE 5. Message Interface Functions

Structure Name

i_msg_t

Synopsis

Structure describing a message, including both the data payload, and related information..

Definition

```
typedef struct i_msg_s {
      struct i_msg_s *next;
      void *msg_handler_struct;
      void *src_handle;
      void *dst_handle;
      i_ubit32_t msg_num;
      void *orig_data;
      i_ubit32_t orig_data_length;
      void *data;
      i_ubit32_t data_length;
      void *response;
      i_ubit32_t response_length;
} i_msg_t;
```

Description of Fields

next	Used for queuing message
msg_handler_struct	Opaque structure
src_handle	Handle of module sending the message
dst_handle	Handle of module the message is being sent to
msg_num	Message number -- this value has other information encoded in it.
orig_data	Pointer to original message data.
orig_data_length	Length of data which *orig_data* points to.
data	Pointer to message data for recipient.
data_length	Length of message data pointed to by *data*.
response	Pointer to response area. This is where a message recipient can write a response.
response length	Length of response data area -- responder must not write more data than the response_length.

Additional Comments

Function:

i_register_message_handler

Prototype:

```
i_status_t
i_register_msg_handler(void *owner_handle,
                       char *owner_name,
                       osm_lockstruct *handler_lock,
                       void (*msg_handler)(void *owner_handle,
                                           i_msg_t *msg));
```

Synopsis

Register a message handler with the message subsystem

Parameters

owner_handle	Pointer to data which will allow the owner of the message handler to associate an incoming message with a particular device or software instance. Personalities would usually use a pointer to their Card_t structure.
owner_name	Unique name which is associated with the *owner_handle*.
handler_lock	Spinlock which should be acquired before the *msg_handler* function is called, and released after the *msg_handler* function is called. If not specified, no lock will be acquired around calls to the *msg_handler.*
msg_handler	Pointer to a function which will handle the messages

Return Value

SND_Success if the message handler was registered successfully, otherwise SND_Fail.

Additional Comments

Any module which wishes to use the core messaging subsystem must register one or more message handlers. Each message handler must have a unique *owner_handle* and *owner_name*.

Function:

i_deregister_msg_handler

Prototype:

```
void
i_deregister_msg_handler(void *owner_handle);
```

Synopsis

Send a message to the message handler associated with the specified handle.

Parameters

owner_handle Handle associated with the message handler which is to be de-registered.

Return Value

None

Additional Comments

As a general rule, only the module which registered the message handler should de-register it.

Function:

i_msg_alloc

Prototype:

```
void *
i_msg_alloc(i_ubit32_t msg_size,
            i_ubit32_t response_size,
            i_ubit32_t flags);
```

Synopsis

Allocate space for message.

Parameters

msg_size	Size of the message to be sent
response_size	Size of the message expected in response to this message
flags	None defined.

Return Value

None

Additional Comments

This is the only function which can be used to allocate memory for messages to be sent with the SND messaging subsystem. “Events”, or messages with no data payload, can be sent without a call to i_msg_alloc().

Examples:

msg_data = i_msg_alloc(512, 0, 0);

> Creates a message 512 bytes long. There is no reponse area, so the same message cannot be used for a response.

msg_data = i_msg_alloc(sizeof(lan_add_mcast_msg_t), sizeof(lan_add_mcast_resp_t), 0);

> Creates a message large enough to hold either lan_add_mcast_t or lan_add_mcast_resp_t. The message will not be large enough to hold both concurrently. The response will be written over the original message.

The caller will receive a pointer to the data payload of the message structure where it should write its message data. Note that although an i_msg_t structure will be created, the caller will not have access to it.

Function:

i_msg_free

Prototype:

```
void
i_msg_free(i_msg_t *msg);
```

Synopsis

Free a message previously allocated with i_msg_alloc.

Parameters

msg The i_msg_t structure allocated by i_msg_alloc().

Return Value

None

Additional Comments

Note that after the call to i_msg_alloc(), the caller does not have the actual i_msg_t structure, and therefore cannot free the message. i_msg_alloc() should not be called unless the caller is sure it intends to send a message.

Function:

i_send_msg

Prototype:

```
void
i_send_msg(void *dst_handle,
           void *src_handle,
           i_ubit32_t msg_num,
           void *msg);
```

Synopsis

Send a message to the message handler associated with the specified handle.

Parameters

dst_handle	Owner handle for destination message handler. This is the handle that the destination message handler was registered with.
src_handle	Owner handle for the source message handler. This is lets the destination know where the message came from. It is also used by the core message subsystem if the message cannot be delivered for some reason.
msg_num	Numerical message value which encodes the type of associated data which *msg* points to. Values for *msg_num* must be constructed with the BLD_MSG_NUM macro.
msg	Pointer to associated data for this message number. If there is no associated data, then NULL should be passed.

Return Value

None

Additional Comments

This is the prefered function to use for sending messages between modules. The sending module must already know the value of dst_handle.

If msg is NULL, then there is no data associated with this message, and it is considered an “event”.

Function:

i_send_msg_byname

Prototype:

```
void
i_send_msg_by_name(char *dst_name,
                   void *src_handle,
                   i_ubit32_t msg_num,
                   void *msg);
```

Synopsis

Send a message to the message handler with the specified name.

Parameters

dst_name	Owner name for destination message handler. This is the name that the destination message handler was registered with.
src_handle	Owner handle for the source message handler. This is lets the destination know where the message came from. It is also used by the core message subsystem if the message cannot be delivered for some reason.
msg_num	Numerical message value which encodes the type of associated data which *msg* points to. Values for *msg_num* must be constructed with the BLD_MSG_NUM macro.
msg	Pointer to associated data for this message number. If there is no associated data, then NULL should be passed.

Return Value

None

Additional Comments

This function should only be used if the destination handle is not known, but the name of the destination message handler is known. The destination handle should be determined as soon as possible, so that i_send_msg can be used.

If msg is NULL, then there is no data associated with this message, and it is considered an “event”.

Function:

i_change_handler_name

Prototype:

```
i_status_t
i_change_handler_name(void *owner_handle,
                      char *new_name);
```

Synopsis

Change the name of a message handler

Parameters

owner_handle Handle associated with the message handler whose name should be changed.

new_name New name for the message handler

Return Value

SND_Success if the name was changed, otherwise SND_Fail.

Additional Comments

Personality Physical Drivers should normally use their SND open firmware path as their message handler name. The Zone is permitted to update the open firmware path after the PPD has set it, usually to append the system device name to the path. When this is done, the Zone should use i_change_handler_name to update the message handler's associated name.

Function:

i_change_handler_lock

Prototype:

```
i_status_t
i_change_handler_lock(void *owner_handle,
                      osm_lockstruct new_lock);
```

Synopsis

Change the associated lock of a message handler

Parameters

owner_handle Handle associated with the message handler whose lock should be changed.

new_lock New lock for the message handler

Return Value

SND_Success if the name was changed, otherwise SND_Fail.

Additional Comments

Certain SND modules may not have a spinlock available at the time they are trying to register their message handlers. An example would be a personality module trying to register a message handler at dev_connect time. These modules can specify a spinlock at a later time using i_change_handler_lock. Another module may also change the spinlock, although this should obviously be done with extreme caution. The intended application of this function is for the Zone to associate the Card lock with the Personalities' message handlers.

Function:

BLD_MSG_NUM

Prototype:

```
i_ubit32_t BLD_MSG_NUM(i_ubit32_t Msg_Class_Idx, i_ubit32_t Msg_Class);
```

Synopsis

Change the associated lock of a message handler

Parameters

Msg_Class_Idx	Message index within the specified message class. Maximum value is 0xffffff.
Msg_Class	General type of message, examples are PERS_LAN_CLASS, IFCON_CLASS, I_MSG_CLASS. Max value is 7f.

Return Value

The resulting message value is returned, which contains theparamater values encoded within it.

Additional Comments

Using bit shifts and logical OR operations, BLD_MSG_NUM combines the arguments in a single 32 bit unsigned integer.

Function:

BLD_RESP_NUM

Prototype:

```
i_ubit32_t BLD_RESP_NUM(i_ubit32_t Msg_Num);
```

Synopsis

Change the associated lock of a message handler

Parameters

Msg_Num Message number of the request that this response is associated with, previously generated with BLD_MSG_NUM.

Return Value

Resulting response message value.

Additional Comments

The original request value is changed so the MSG_RESPONSE bit is set.

Messages used by messaging subsystem

Message Name

I_MSG_ERROR

Synopsis

An error prevented a message from being delivered.

Message Data

```
typedef struct i_msg_error_s {
    void *dst_handle;
    msg_status_t msg_status;
    i_ubit32_t   orig_msg_num;
    void *orig_msg_contents;
} i_msg_error_t;
```

Message Fields

dst_handle	Intended destination of message.
msg_status	Reason the message could not be delivered.
orig_msg_num	Message number of the undeliverable message.
orig_msg_contents	Contents of the undeliverable message

Response Name

None

Response Data

None

Additional Comments

Because this message comes directly from the message subsystem, and bypasses the message queue, the receiver must not respond with another message, since this could result in a recursive loop and stack overflow.

5.3 Ifcon Interface

Function Name	Synopsis
ifcon_connect	Get ifcon handle for use with ifcon state machine and other ifcon services.
ifcon_disconnect	Free ifcon handle previously allocated with ifcon_connect.
ifcon_stack_con	Move interface to “stack connected” state from the “device connected” state.
ifcon_stack_discon	Move interface to “device connected” state, which can also be thought of as “stack disconnected”.
ifcon_enable	Move interface to “enabled” state from either the “stack connected” state or the “device connected” state.
ifcon_disable	Move interface to “stack connected” state from the “enabled” state.
ifcon_reset	Reset the hardware and driver of a device in the “enabled” state. Maintains interface configuration, including multicast address and promiscuous mode state.
ifcon_reset_clean	Resets the hardware and driver of a device in the “enabled” state. Interface configuration, including multicast addressing and promiscuous state are return to power-on values.
ifcon_standalone_diags	Run detailed diagnostics (if available) on and interface in the “device connected” state.
ifcon_sendpkt	Macro to send a packet, given an ifcon_info_t structure and a buffer.
ifcon_scsicmd	Macro to start a SCSI cmd, given an ifcon_info_t structure and a buffer.

TABLE 6. State Machine Function Interface

Function:

ifcon_connect

Prototype:

```
void*ifcon_connect(void *snif, ifcon_info_t *ifcon_info,
                   char *if_name);
```

Synopsis

Get ifcon handle for use with ifcon state machine and other ifcon services.

Parameters

snif	Handle that the SIF or NIF associates with this interface
ifcon_info	Pointer to a structure allocated by the SIF or NIF, which will be maintained by the ifcon services. This is used to allow ifcon and S/NIF instant access to certain pieces of information, without requiring the exchange of messages. Certain portions of the structure must be completed by the S/NIF before calling ifcon_connect.
if_name	String containing a unique name for the interface the ifcon will be managing. Ifcon will prepend “ifcon_” to this name, and use it as the name for its message handler for this interface.

Return Value

Opaque handle for use with ifcon services.

Additional Comments

An “ifcon” handle must be obtained by any NIF or SIF level interface that needs to use ifcon services. These services include the ifcon initialization state machine, multicast address management, and promiscuous mode state tracking.

Function:

ifcon_disconnect

Prototype:

```
void ifcon_disconnect(ifcon_t *ifcon);
```

Synopsis

Free ifcon handle previously allocated with ifcon_connect.

Parameters

ifcon Ifcon handle previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

Free the ifcon handle, along with any associated resources.

Function:

ifcon_stack_con

Prototype:

```
void ifcon_stack_con(ifcon_t *ifcon);
```

Synopsis

Move interface to “stack connected” state from the “device connected” state.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

See IFCON_STATE_CHANGE_ALERT message description.

Function:

ifcon_stack_discon

Prototype:

```
void ifcon_stack_discon(ifcon_t *ifcon);
```

Synopsis

Move interface to “device connected” state, which can also be thought of as “stack disconnected”.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

See IFCON_STATE_CHANGE_ALERT message description.

Function:

ifcon_enable

Prototype:

```
void ifcon_enable(ifcon_t *ifcon);
```

Synopsis

Move interface to “enabled” state from either the “stack connected” state or the “device connected” state.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

See IFCON_STATE_CHANGE_ALERT message description.

Function:

ifcon_disable

Prototype:

```
void ifcon_disable(ifcon_t *ifcon);
```

Synopsis

Move interface to “stack connected” state from the “enabled” state.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

See IFCON_STATE_CHANGE_ALERT message description.

Function:

ifcon_reset

Prototype:

```
void ifcon_reset(ifcon_t *ifcon);
```

Synopsis

Reset the hardware and driver of a device in the “enabled” state. Maintains interface configuration, including multicast address and promiscuous mode state.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

Causes a disable operation, followed by an enable operation.

Function:

ifcon_reset_clean

Prototype:

```
void ifcon_reset_clean(ifcon_t *ifcon);
```

Synopsis

Resets the hardware and driver of a device in the “enabled” state. Interface configuration, including multicast addressing and promiscuous state are return to power-on values.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

Causes a disable operation, followed by an enable operation, but state is not maintained.

Function:

ifcon_standalone_diags

Prototype:

```
i_status_t ifcon_standalone_diags(ifcon_t *ifcon,
                                  i_ubit32_t diag_level);
```

Synopsis

Run detailed diagnostics (if available) on and interface in the “device connected” state.

Parameters

ifcon Ifcon handle for the desired interface, previously acquired through a call to ifcon_connect.

Return Value

None

Additional Comments

See IFCON_STATE_CHANGE_ALERT message description.

Function:

ifcon_sendpkt

Prototype:

```
i_status_t ifcon_sendpkt(ifcon_info_t ifcon_info, bdm_buf_t *buf);
```

Synopsis

Macro to send a packet given an ifcon_info_t structure, and a buffer.

Parameters

ifcon_info ifcon_info_t structure. Not a pointer!

buf The bdm_buf_t structure corresponding to the packet to be sent.

Return Value

None

Additional Comments

Since this is a macro, there is no performance impact to passing parameters “by value”.

Function:

ifcon_scsicmd

Prototype:

```
i_status_t ifcon_scsicmd(ifcon_info_t ifcon_info, bdm_buf_t *buf);
```

Synopsis

Macro to start a SCSI cmd, given an ifcon_info_t structure and a buffer.

Parameters

ifcon_info ifcon_info_t structure. Not a pointer!

buf The bdm_buf_t structure corresponding to the SCSI command.

Return Value

None

Additional Comments

Since this is a macro, there is no performance impact to passing parameters “by value”.

Messages used by Ifcon

Message Name

IFCON_STATE_CHANGE_ALERT

Synopsis

Indicates a state change on a device being controlled by the ifcon.

Message Data

```
typedef struct {
    ifcon_state_t current_state;
    i_boolean_t   complete;
} ifcon_state_change_alert_t;
```

Message Fields

current_state The current state of the interface referred to by this ifcon handle (src_handle)

complete If complete is TRUE, then the current_state is the state that was requested. For example, if the Zone called ifcon_enable(), then at some point later it would get a message with current_state=Enabled and complete=TRUE.

Response Name

None

Response Data

None

Additional Comments

Operations performed in response to state change alerts should be carefully thought out. In general it is a good policy to not do anything (except to update Zone state information) unless complete=TRUE.

5.4 Memory Allocation Interface (i_mem)

Function Name	Synopsis
i_mem_alloc	Allocate the specified number of bytes from the specified memory pool
i_mem_zalloc	Allocate and zero the specified number of bytes from the specified memory pool
i_mem_free	Free a portion of memory previously allocated by i_mem_alloc() or i_mem_zalloc() to the specified pool
i_mem_free_any	Free a previously allocated portion of memory to whatever pool it was allocated from. This function has more overhead than i_mem_free().
i_mem_initpool	Create a pool of memory for use with i_mem_alloc(), etc.
i_mem_freepool	Release a pool of memory previously allocated with i_mem_initpool().
i_mem_usage_report	Print a usage report for the existing memory pools.

TABLE 7. Memory Allocation Function Interface

Function:

i_mem_alloc

Prototype:

```
void *i_mem_alloc(i_mempoolp pool_type, i_ubit32_t nbytes);
```

Synopsis

Allocate the specified number of bytes from the specified memory pool.

Parameters

pool_type	Memory pool handle previously allocated with i_mem_initpool.
nbytes	Number of bytes requested.

Return Value

Pointer to the allocated memory. If the memory could not be allocated, then NULL will be returned.

Additional Comments

None

Function:

i_mem_zalloc

Prototype:

```
void *i_mem_zalloc(i_mempoolp pool_type, i_ubit32_t nbytes);
```

Synopsis

Allocate and zero the specified number of bytes from the specified memory pool.

Parameters

pool_type	Memory pool handle previously allocated with i_mem_initpool.
nbytes	Number of bytes requested.

Return Value

Pointer to the allocated memory. The allocated memory will be initialized with zeros. If the memory could not be allocated, then NULL will be returned.

Additional Comments

None

Function:

i_mem_free

Prototype:

```
void  i_mem_free(i_mempoolp pool_type, void *memptr);
```

Synopsis

Free a portion of memory previously allocated by i_mem_alloc() or i_mem_zalloc() to the specified pool.

Parameters

pool_type	Memory pool handle previously allocated with i_mem_initpool.
memptr	Pointer to memory region previously allocated with i_mem_alloc or i_mem_zalloc.

Return Value

None

Additional Comments

None

Function:

i_mem_free_any

Prototype:

```
void i_mem_free_any(void *memptr);
```

Synopsis

Free a previously allocated portion of memory to whatever pool it was allocated from. This function has more overhead than i_mem_free().

Parameters

memptr Pointer to memory region previously allocated with i_mem_alloc or i_mem_zalloc.

Return Value

None

Additional Comments

None

Function:

i_mem_initpool

Prototype:

```
i_mempoolp i_mem_initpool(char *pool_name, i_ubit32_t instance,
                          i_ubit32_t min_step, i_ubit32_t keep_bytes,
                          i_ubit32_t max_bytes);
```

Synopsis

Create a pool of memory for use with i_mem_alloc(), etc.

Parameters

pool_name	User specified pool name, max length is 64 bytes.
instance	Instance number for this pool. Can be used to differentiate a set of pools which are used for the same basic purpose. For example, a personality module might create a separate memory pool for each hardware interface it controls, and use the same name, providing different instance numbers.
min_step	Minimum step size for pieces of allocated memory. This also determines the alignment of allocated memory.
keep_bytes	Number of bytes to be kept in this memory pool when memory is freed to the pool. If the pool has this much memory already which memory is freed to it, then the new memory will be freed to the OS instead.
max_bytes	Maximum number of bytes that can be allocated from the OS for this pool. If the current size (user allocated and in-pool free) is less than this amount, new requests that can’t be satisfied from the pool will cause an OS allocation rquest. Otherwise, the allocation will fail.

Return Value

Opaque memory pool handle.

Additional Comments

None

Function:

i_mem_freepool

Prototype:

```
void i_mem_freepool(i_mempoolp pool_type);
```

Synopsis

Release a pool of memory previously allocated with i_mem_initpool().

Parameters

pool_type Memory pool handle previously allocated with i_mem_initpool.

Return Value

None

Additional Comments

None

Function:

i_mem_usage_report

Prototype:

```
void i_mem_usage_report();
```

Synopsis

Print a usage report for the existing memory pools.

Parameters

None

Return Value

None

Additional Comments

This function provides a detailed memory usage report of all the existing memory pools. The report is printed to the same device that debug messages are printed to.

5.5 String and Memory Operations

Function Name	Synopsis
i_printf	Replacement for printf()
i_sprintf	Replacement for sprintf().
i_vsprintf	Replacement for vsprintf().
i_strcat	Replacement for strcat().
i_strspn	Replacement for strspn().
i_strcspn	Replacement for strcspn().
i_strlen	Replacement for strlen().
i_strcat	Replacement for strcat().
i_dbgmsg	Identical to i_printf, but suppresses output unless Core code is compiled with -DDEBUG.
I_DBGMSG	Print a debug message with filtering
i_strin	Determine if a given string is contained within another given string
i_ston_d	Convert a decimal ASCII representation of a number to an integer.
i_ston_h	Convert a hex ASCII representation of a number to an integer.
i_addr_sprintf	Create a string representation of a six byte MAC address.
i_bprintf	Memory dump in hex bytes with ascii equivelants.
i_sbprintf	Same as i_bprintf, but output is placed in a string.
i_bcopy	Copy memory from one location to another
i_bzero	Zero a region of memory
i_bcmp	Compare one region of memory to another
I_HTOB32	Convert 32 bit value from host byte ordering to big endian format.
I_HTOB16	Convert 16 bit value from host byte ordering to big endian format.
I_BTOH32	Convert 32 bit value from big endian format to host byte ordering.
I_BTOH16	Convert 16 bit value from big endian format to host byte ordering.
I_HTOL32	Convert 32 bit value from host byte ordering to little endian format.
I_HTOL16	Convert 16 bit value from host byte ordering to little endian format.
I_LTOH32	Convert 32 bit value from little endian format to host byte ordering.
I_LTOH16	Convert 16 bit value from little endian format to host byte ordering.
IC_HTOB32	Convert 32 bit constant from host byte ordering to big endian format.
IC_HTOB16	Convert 16 bit constant from host byte ordering to big endian format.
IC_BTOH32	Convert 32 bit constant from big endian format to host byte ordering.
IC_BTOH16	Convert 16 bit constant from big endian format to host byte ordering.
IC_HTOL32	Convert 32 bit constant from host byte ordering to little endian format.
IC_HTOL16	Convert 16 bit constant from host byte ordering to little endian format.
IC_LTOH32	Convert 32 bit constant from little endian format to host byte ordering.
IC_LTOH16	Convert 16 bit constant from little endian format to host byte ordering.

TABLE 8. String and Memory Operations

Function:

i_dbgmsg

Prototype:

```
void i_dbgmsg(char *fmt, ...);
```

Synopsis

Identical to i_printf, but suppresses output unless Core code is compiled with -DDEBUG.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

I_DBGMSG

Prototype:

```
void I_DBGMSG(i_ubit32_t debug_level, char *fmt, ...);
```

Synopsis

Print a debug message with filtering.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_strin

Prototype:

```
i_ubit32_t i_strin(char *srch_from, char *srch_for);
```

Synopsis

Determine if a given string is contained within another given string.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_ston_d

Prototype:

```
i_ubit32_t i_ston_d(char *strnum, char **rstrptr);
```

Synopsis

Convert a decimal ASCII representation of a number to an integer.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_ston_h

Prototype:

```
i_ubit32_t i_ston_h(char *strnum, char **rstrptr);
```

Synopsis

Convert a hex ASCII representation of a number to an integer.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_addr_sprintf

Prototype:

```
i_ubit8_t *i_addr_sprintf(i_ubit8_t *buf, i_ubit8_t *addr,
                          i_ubit32_t length, char separater);
```

Synopsis

Create a string representation of a arbitrary length address.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_bprintf

Prototype:

```
i_ubit32_t i_bprintf(unsigned char *abuf, i_ubit32_t size);
```

Synopsis

Memory dump in hex bytes with ascii equivelants.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_sbprintf

Prototype:

```
char *i_sbprintf(char *s, unsigned char *abuf, i_ubit32_t size);
```

Synopsis

Same as i_bprintf, but output is placed in a string.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_bcopy

Prototype:

```
void i_bcopy(char *)src,(char *)dst, i_ubit32_t len);
```

Synopsis

Copy memory from one location to another

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_bzero

Prototype:

```
void i_bzero(char *start, i_ubit32_t length);
```

Synopsis

Zero a region of memory.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

i_bcmp

Prototype:

```
i_bcmp(char *s1, char *s2, i_ubit32_t length);
```

Synopsis

Compare one region of memory to another.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

I_HTOB32, I_HTOB16, I_BTOH32, I_BTOH16, I_HTOL32, I_HTOL16, I_LTOH32, I_LTOH16, IC_HTOB32, IC_HTOB16, IC_BTOH32, IC_BTOH16, IC_HTOL32, IC_HTOL16, IC_LTOH32, IC_LTOH16

Prototype:

```
i_ubit32_t I_HTOB32(i_ubit32_t val);
i_ubit16_t I_HTOB32(i_ubit16_t val);
i_ubit32_t I_BTOH32(i_ubit32_t val);
i_ubit16_t I_BTOH16(i_ubit16_t val);

i_ubit32_t I_HTOL32(i_ubit32_t val);
i_ubit16_t I_HTOL32(i_ubit16_t val);
i_ubit32_t I_LTOH32(i_ubit32_t val);
i_ubit16_t I_LTOH16(i_ubit16_t val);

i_ubit32_t IC_HTOB32(i_ubit32_t val);
i_ubit16_t IC_HTOB32(i_ubit16_t val);
i_ubit32_t IC_BTOH32(i_ubit32_t val);
i_ubit16_t IC_BTOH16(i_ubit16_t val);

i_ubit32_t IC_HTOL32(i_ubit32_t val);
i_ubit16_t IC_HTOL32(i_ubit16_t val);
i_ubit32_t IC_LTOH32(i_ubit32_t val);
i_ubit16_t IC_LTOH16(i_ubit16_t val);
```

Synopsis

Convert between host byte ordering and big or little endian byte ordering.

Parameters

val The value to be converted

Return Value

Converted value.

Additional Comments

I_HTOB32, I_HTOB16, I_BTOH32, I_BTOH16, I_HTOL32, I_HTOL16, I_LTOH32, I_LTOH16

These macros should be used to convert values which are variable and cannot be computed at compile time.

IC_HTOB32, IC_HTOB16, IC_BTOH32, IC_BTOH16, IC_HTOL32, IC_HTOL16, IC_LTOH32, IC_LTOH16

These macros should be used to convert constant values which are evaluated at compile time. The byte swapping will also be performed at compile time.

5.6 Locking Interface

Function Name	Synopsis
i_create_lock	Allocate and initialize a lock structure
i_free_lock	Free a lock structure
i_lock	Acquire a lock, previously allocated with i_create_lock()
i_unlock	Release a lock, previously acquired with i_lock().
i_plock	For use by personalities. Acquire a lock, previously allocated with i_create_lock().
i_punlock	For use by personalities. Release a lock, previously acquired with i_lock().

TABLE 9. Lock Interface Functions

Function:

i_create_lock

Prototype:

```
i_lock_t *i_create_lock(char *name, i_boolean_t prevent_ints,
                        void *bus_info);
```

Synopsis

Allocate and initialize a lock structure.

Parameters

name	Descriptions associated with this lock
prevent_ints	TRUE if interrupts should be disabled while the lock is held, otherwise false.
bus_info	Pointer to a bus structure containing data which would be used by the OSM to implement the prvent_ints function. In the SND enviroment, a pointer to a bus_t structure should be passed here.

Return Value

A pointer to an i_lock_t structure if successfull, otherwise NULL.

Additional Comments

i_create_lock, i_lock, and i_unlock sit on top of the primitive osm_i_createlock, OSM_I_LOCK, and OSM_I_UNLOCK services, providing additional statistics and debug information in a portable fashion, without requiring each OSM to implement the debug and statistics code.

Function:

i_free_lock

Prototype:

```
void i_free_lock(i_lock_t *lock);
```

Synopsis

Free a lock structure previously allocated with i_create_lock.

Parameters

lock Pointer to a lock structure previously allocated with i_create_lock.

Return Value

None.

Additional Comments

None

Function:

i_lock, i_plock

Prototype:

```
void i_lock(i_lock_t *lock);
void i_plock(i_lock_t *lock);
```

Synopsis

Acquire a lock, previously allocated with i_create_lock().

Parameters

Return Value

TBD

Additional Comments

TBD

Function:

i_unlock, i_punlock

Prototype:

```
void i_unlock(i_lock_t *lock);
void i_punlock(i_lock_t *lock);
```

Synopsis

Release a lock, previously acquired with i_lock()/i_plock().

Parameterss

Return Value

TBD

Additional Comments

TBD

5.7 Queuing Interface

Function Name	Synopsis
INSQH_SLL G_INSQH_SLL	Insert an element before the head of a singly linked list.
INSQT_SLL G_INSQT_SLL	Insert an element after the tail of a singly linked list
INSQ_SLL G_INSQ_SLL,	Insert an element after a specified element of a singly linked list (do not use in performance path).
REMQH_SLL G_REMQH_SLL	Remove the element at the head of a singly linked list.
REMQT_SLL G_REMQT_SLL	Remove the element at the tail of a singly linked list (do not use in performance path).
REMQ_SLL, G_REMQ_SLL	Remove a given element from a singly linked list (do not use in performance path).
INSQH_DLL, G_INSQH_DLL	Insert an element before the head of a singly linked list.
INSQT_DLL, G_INSQT_DLL	Insert an element after the tail of a singly linked list
INSQ_DLL, G_INSQ_DLL	Insert an element after a specified element of a singly linked list.
REMQH_DLL, G_REMQH_DLL	Remove the element at the head of a singly linked list
REMQT_DLL, G_REMQT_DLL	Remove the element at the tail of a singly linked list
REMQ_DLL, G_REMQ_DLL	Remove a given element from a singly linked list
INSQHEAD	OBSOLETE: Insert an element before the head of a singly linked list
INSQTAIL	OBSOLETE: Insert an element after the tail of a singly linked list
REMQHEAD	OBSOLETE: Remove the element at the head of a singly linked list
REMQTAIL	OBSOLETE: Remove the element at the tail of a singly linked list
QINIT	Initialize an i_queue_t structure
QHEAD	Returns the head of a linked list
QTAIL	Returns the tail of a linked list
QNEXT	Returns the next element after a specified element in a linked list
QSIZE	Returns the number of elements in a linked list
QFOREACH, G_QFOREACH	Performs a specified operation once on each element of a linked list

TABLE 10. Queue Interface Functions

Function:

INSQH_SLL, G_INSQH_SLL

Prototype:

```
void INSQH_SLL(Queue, NewElement, ElementType);

void G_INSQH_SLL(Queue, NewElement, NextFieldName, ElementType);
```

Synopsis

Insert an element before the head of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQT_SLL, G_INSQT_SLL

Prototype:

```
void INSQT_SLL(Queue, NewElement, ElementType);

void G_INSQT_SLL(Queue, NewElement, NextFieldName, ElementType);
```

Synopsis

Insert an element after the tail of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQ_SLL, G_INSQ_SLL

Prototype:

```
void INSQ_SLL(Queue, NewElement, AfterElement, ElementType);

void G_INSQ_SLL(Queue, NewElement, AfterElement, NextFieldName,
                ElementType);
```

Synopsis

Insert an element after a specified element of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQH_SLL, G_REMQH_SLL

Prototype:

```
void REMQH_SLL(Queue, ElementVar, ElementType);

void G_REMQH_SLL(Queue, ElementVar, NextFieldName, ElementType);
```

Synopsis

Remove the element at the head of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQT_SLL, G_REMQT_SLL

Prototype:

```
void REMQT_SLL(Queue, ElementVar, ElementType);

void G_REMQT_SLL(Queue, ElementVar, NextFieldName, ElementType);
```

Synopsis

Remove the element at the tail of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQ_SLL, G_REMQ_SLL

Prototype:

```
void REMQ_SLL(Queue, Element, ElementType);

void G_REMQ_SLL(Queue, Element, NextFieldName, ElementType);
```

Synopsis

Remove a given element from a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQH_DLL, G_INSQH_DLL

Prototype:

```
void INSQH_DLL(Queue, NewElement, ElementType);

void G_INSQH_DLL(Queue, NewElement, NextFieldName, PrevFieldName,
                 ElementType);
```

Synopsis

Insert an element before the head of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQT_DLL, G_INSQT_DLL

Prototype:

```
void INSQT_DLL(Queue, NewElement, ElementType);

void G_INSQT_DLL(Queue, NewElement, NextFieldName, PrevFieldName,
                 ElementType);
```

Synopsis

Insert an element after the tail of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQ_DLL, G_INSQ_DLL

Prototype:

```
void INSQ_DLL(Queue, NewElement, AfterElement, ElementType);

void G_INSQ_DLL(Queue, NewElement, AfterElement,
                NextFieldName, PrevFieldName,
                ElementType);
```

Synopsis

Insert an element after a specified element of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQH_DLL, G_REMQH_DLL

Prototype:

```
void REMQH_DLL(Queue, ElementVar, ElementType);

void G_REMQH_DLL(Queue, ElementVar, NextFieldName, PrevFieldName,
                 ElementType);
```

Synopsis

Remove the element at the head of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQT_DLL, G_REMQT_DLL

Prototype:

```
void REMQT_DLL(Queue, ElementVar, ElementType);

void G_REMQT_DLL(Queue, ElementVar, NextFieldName, ElementType);
```

Synopsis

Remove the element at the tail of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQ_DLL, G_REMQ_DLL

Prototype:

```
void REMQ_DLL(Queue, Element, ElementType);

void G_REMQ_DLL(Queue, Element, NextFieldName, ElementType);
```

Synopsis

Remove a given element from a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQHEAD

Prototype:

```
void INSQHEAD(Queue, NewElement, NextFieldName, ElementType);
```

Synopsis

OBSOLETE: Insert an element before the head of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

INSQTAIL

Prototype:

```
void INSQTAIL(Queue, NewElement, NextFieldName, ElementType);
```

Synopsis

OBSOLETE: Insert an element after the tail of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQHEAD

Prototype:

```
void REMQHEAD(Queue, ElementVar, NextFieldName, ElementType);
```

Synopsis

OBSOLETE: Remove the element at the head of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

REMQTAIL

Prototype:

```
void REMQTAIL(Queue, ElementVar, NextFieldName, ElementType);
```

Synopsis

OBSOLETE: Remove the element at the tail of a singly linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

QINIT

Prototype:

```
void QINIT(Queue);
```

Synopsis

Returns the head of a linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

QHEAD

Prototype:

```
Element *QHEAD(Queue);
```

Synopsis

Returns the head of a linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

QTAIL

Prototype:

```
Element *QTAIL(Queue);
```

Synopsis

Returns the tail of a linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

QNEXT

Prototype:

```
Element *QNEXT(Queue, Element, NextFieldName);
```

Synopsis

Returns the next element after a specified element in a linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

QSIZE

Prototype:

```
i_ubit32_t QSIZE(Queue);
```

Synopsis

Returns the number of elements in a linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

QFOREACH, G_QFOREACH

Prototype:

```
void QFOREACH(Queue, Element, ElementType);

void G_QFOREACH(Queue, Element, NextFieldName, ElementType);
```

Synopsis

Performs a specified operation once on each element of a linked list.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

5.8 Timer Interface

Function Name	Synopsis
i_timer_connect	Create a timer pool which can be used to request timeouts.
i_timer_disconnect	Free a timer pool previously allocated with i_timer_connect.
i_timeout	Call a timeout function after a specified interval has passed.
i_timeout_cancel	Cancel a timeout previously scheduled with i_timeout().

TABLE 11. Time Interface Functions

Function:

i_timer_connect

Prototype:

```
i_tpool_t *i_timer_connect(void *bus_info);
```

Synopsis

Create a timer pool which can be used to request timeouts.

Parameters

bus_info Pointer to bus_information which the OSM will be able to use to create the spinlock and lock out hardware. If the timers need not be synchronized against interrupts, then the bus_info field can be NULL.

Return Value

Pointer to a timer pool structure

Additional Comments

Users of i_timeout must first call i_timer_connect in order to obtain an i_tpool_t structure.

Function:

i_timer_disconnect

Prototype:

```
void i_timer_disconnect(i_tpool_t *tpool);
```

Synopsis

Free a timer pool previously allocated with i_timer_connect.

Parameters

tpool Timer pool previously allocated with i_timer_connect.

Return Value

None

Additional Comments

Frees the i_tpool_t structure, and any associated resources.

Function:

i_timeout

Prototype:

```
i_status_t i_timeout(i_tpool_t *tpool, i_thandle_t *thandle,
                     tfunc trtn, void *targ,
                     i_ubit32_t tval_msecs);
```

Synopsis

Call a timeout function after a specified interval has passed.

Parameters

tpool	Timer pool previously allocated with i_timer_connect.
thandle	Users of i_timeout allocate this structure, but do not need to fill it in.
trtn	Function to call when the timeout expires.
targ	Argument to pass to the timeout function (trtn) if the timer expires.
tval_msecs	Number of milliseconds to wait before the timer fires.

Return Value

iSuccess if timer was started, otherwise iFail.

Additional Comments

None.

Function:

i_timeout_cancel

Prototype:

```
void i_timeout_cancel(i_thandle_t *thandle);
```

Synopsis

Cancel a timeout previously scheduled with i_timeout().

Parameters

thandle Timer handle previously used in i_timeout to start a timeout.

Return Value

None

Additional Comments

If the timeout already fired, this function will have no effect.

5.9 Other

Function Name	Synopsis
snd_connect_downward	Iterates through list of buses and personality modules, hunting for a match for a particular device.
snd_get_pvd_instances	For given PVD, find all its virtual device instances.
snd_init_card_struct	Fills in many of the fields of a Card_t structure.
snd_build_OF_path	Build an SND open firmware name string.
snd_create_pregion_lock	Create a personality region lock for a given device instance.
snd_free_pregion_lock	Free a personality region lock previosly allocated with snd_create_pregion_lock().
SND_LOCK_PREGION_C	Acquire a personality region lock for a given device instance, give the devices corresponding Card_t structure.
SND_UNLOCK_PREGION_C	Free a personality region lock given the devices Card_t structure.
SND_LOCK_PREGION_B	Acquire a personality region lock for a given device instance, give the system bus_t structure for the device.
SND_UNLOCK_PREGION_B	Free a personality region lock given the system bus_t structure for the device.

TABLE 12. Other Core Interface Functions

Function:

snd_connect_downward

Prototype:

```
pers_dev_t *
snd_connect_downward(bus_t *parent_bus,
                     snd_dev_id_t id_val);
```

Synopsis

Iterates through list of buses and personality modules, hunting for a match for a particular device.

Parameters

Return Value

Additional Comments

Function:

snd_get_pvd_instances

Prototype:

```
pers_dev_t *
snd_get_pvd_instances(char *pvd_short_name);
```

Synopsis

For given PVD, find all its virtual device instances.

Parameters

Return Value

Additional Comments

Function:

snd_init_card_structure

Prototype:

```
pers_dev_t *
snd_init_card_struct(i_ubit32_t sizeof_Card_t,
                     pers_props_t *pers_props_ptr,
                     bus_t *pBUS);
```

Synopsis

Fills in many of the fields of a Card_t structure.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

snd_build_OF_path

Prototype:

```
void snd_build_OF_path(Card_t *pC, char *card_path_name);
```

Synopsis

Build an SND open firmware name string.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

snd_create_pregion_lock

Prototype:

```
void snd_create_pregion_lock(Card_t *pC);
```

Synopsis

Create a personality region lock for a given device instance.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

snd_free_pregion_lock

Prototype:

```
void snd_free_pregion_lock(Card_t *pC);
```

Synopsis

Free a personality region lock previosly allocated with snd_create_pregion_lock().

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

SND_LOCK_PREGION_C

Prototype:

```
void SND_LOCK_PREGION_C(Card_t *pC);
```

Synopsis

Acquire a personality region lock for a given device instance, give the devices corresponding Card_t structure.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

SND_UNLOCK_PREGION_C

Prototype:

```
void SND_UNLOCK_PREGION_C(Card_t *pC);
```

Synopsis

Free a personality region lock given the devices Card_t structure.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

SND_LOCK_PREGION_B

Prototype:

```
void SND_LOCK_PREGION_B(bus_t *pBUS);
```

Synopsis

Acquire a personality region lock for a given device instance, give the system bus_t structure for the device.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

Function:

SND_UNLOCK_PREGION_B

Prototype:

```
void SND_UNLOCK_PREGION_B(bus_t *pBUS);
```

Synopsis

Free a personality region lock given the system bus_t structure for the device.

Parameters

TBD

Return Value

TBD

Additional Comments

TBD

6.0 Personality Interface Reference

6.1 General Description

The interface to an SND personality module consists of a set of function entry points, and a set of messages the personality can send and receive. Throughout the Personality Interface Reference, function entry points will have the symbol <Pers> embedded in there names. In an actual personality module, this symbol would be replaced with the module's name. For example, the <Pers>_dev_connect and <Pers>_dev_disconnect entry points for the motofsi module would be named motofsi_dev_connect and motofsi_dev_disconnect.

Every Personality must have a "load" entry point, called <Pers>_load, which calls oim_add_pers() to install itself. This makes the SND environment aware of the personality, which in turn allows SND to offer devices to the Personality. Two functions are exported through the card_mod_props structure: <Pers>_dev_connect and <Pers>_dev_disconnect. This portion of the interface to a personality will be refered to as the "device connection interface." The "message interface" refers to the interface defined by the messages the personality is capable of receiving, and the messages the personality is capable of sending. "Performance Interface" is the interface for sending packets or satisfying SCSI requests, specified within a LAN_STACK_CONN_RESP or SCSI_STACK_CONN_RESP message. "Interrupt Interface" is the interface for handling interrupts from the adapter, specified within a call to bus_add_isr().

6.2 Device Connection Interface

Function Name	Synopsis
<Pers>_load	Called at load time, at which point the personality installs itself into the SND environment
<Pers>_dev_connect	Determine if an offered device should be controlled by this personality, or probe for devices controlled by this personality. Create a control structure for each logical device it controls
<Pers>_dev_disconnect	Release all remaining resources associated with a particular logical device, and then free the control structure related to that device.
<Pers>_unload	Release all resources not associated with a particular device, and uninstall from SND environment.

TABLE 13. Device Connection Interface Entry Points

Function:

<Pers>_load

Prototype:

```
void <Pers>_load(void)
```

Synopsis

Called at load time, at which point the personality installs itself into the SND environment

Parameters

None

Return Value

None

Detailed Description

The <Pers>_load function should call oim_add_pers to register the personality with the OIM. Personalities which support multiple bus types must have separate pers_props_t structures for each bus type, and oim_add_pers must be called once per pers_props_t structure.

Function:

<Pers>_dev_connect

Prototype:

```
pers_dev_t *<Pers>_dev_connect(bus_t *pbus, snd_dev_id_t *dev_id);
```

Synopsis

Determine if an offered device should be controlled by this personality, or probe for devices controlled by this personality. Create a control structure for each logical device it controls

Parameters

pbus	Pointer to a bus_t structure created by the parent bus of the offered device
dev_id	ID structure which contains the appropriate information to identify the offered device on the parent bus. See the description of snd_dev_id_t for more details.

Return Value

Pointer to a pers_dev_t structure or list of pers_dev_t structure (one per device created). Returning NULL indicates either the device was not recognized, no devices were found, the device could not be initialized.

Detailed Description

Personality Physical Drivers:

On buses which can be enumerated, such as PCI, EISA, and SBus, a bus module will call <Pers>_dev_connect with the *dev_id* for each device encountered on that bus. If the device was not recognized, NULL will be returned. If an error occured initializing the device, NULL will be returned.

Buses which cannot be enumerated, such as VME and ISA, should not pass a *dev_id* value. For each personality associated with the bus, that personalities <Pers>_dev_connect function should be called exactly once. The <Pers>_dev_connect will probe for supported devices, and return a list of pers_dev_t structures corresponding to the devices which were discovered. If no devices were discovered (or if no devices could be intialized), NULL will be returned.

Personality Virtual Drivers

TBD

Personality Service Drivers

TBD

Function:

<Pers>_dev_disconnect

Prototype:

```
void <Pers>_dev_disconnect(pers_dev_t *pC);
```

Synopsis

Free the remaining resources associated with a particular logical device

Parameters

pC Pointer to a single pers_dev_t structure previous returned by a call to <Pers>_dev_connect.

Return Value

None

Detailed Description

<Pers>_dev_disconnect frees the pers_dev_t structure and any remaining associated resources, generally in preparation for an unload.

Function:

<Pers>_unload

Prototype:

```
void <Pers>_unload(void)
```

Synopsis

Free any remaining resources not associated with pers_dev_t structures

Parameters

None

Return Value

None

Detailed Description

Before <Pers>_unload is called, <Pers>_dev_disconnect should have been called once for each pers_dev_t which had been created by the personality module. <Pers>_unload is then responsible for freeing any remaining resources.

<Pers>_unload is not responsible for freeing resources associated with pers_dev_t structures.

6.3 Performance Interface

Although <Pers>_sendpkt and <Pers>_scsi_cmd are both listed below, personalities will generally only implement one or the other depending on whether the personality is a network or storage driver, respectively. The NIF or SIF receives these entry points from the LAN_STACK_CON_RESP or SCSI_STACK_CON_RESP message. The naming convention below is recommended, but not required.

Function Name	Synopsis
<Pers>_sendpkt	Send a network packet, prepared by the NIF.
<Pers>_scsi_cmd	Perform the requested SCSI operation, prepared by the SIF.

TABLE 14. Performance Interface Entry Points

Function:

<Pers>_sendpkt

Prototype:

```
i_status_t <Pers>_sendpkt(pers_dev_t *pC, bdm_buf_t *buf);
```

Synopsis

Transmit a network packet

Parameters

pC Pointer to the pers_dev_t structure associated with the interface on which the network packet should be transmitted

buf Buffer structure containing the packet data to be transmitted, along with any upper layer headers.

Return Value

If the packet was successfully transmitted or queued to the hardware, or if the device is not in the correct state to transmit the packet, SND_Success will be returned. If there was an error transmitting or queuing the packet.

Detailed Description

Before calling <Pers>_sendpkt, the packet buffer must contain all the packet data, including standardized network headers such as ethernet or FDDI MAC headers. Space must be provide for hardware specific header data if the personality specified an extra_tx_hdr_bytes value during the stack connect phase.

NIF’s are expected to build valid ethernet and FDDI headers, but are not expected to build Fibre Channel headers. Therefore, Fibre Channel networking personalities will need to build the FC headers themselves.

Function:

<Pers>_scsi_cmd

Prototype:

```
i_status_t <Pers>_scsi_cmd(pers_dev_t *pC, bdm_buf_t *buf);
```

Synopsis

Perform the SCSI command described by *buf*.

Parameters

pC Pointer to the pers_dev_t structure associated with the interface on which the SCSI command should be transmitted

buf Buffer structure containing the SCSI command to be transmitted, along with any necessary data buffers.

Return Value

If the SCSI command was successfully transmitted or queued to the hardware, SND_Success or SND_Pending should be returned. If there was an error transmitted or queuing the command, SND_Fail should be returned.

Detailed Description

TBD

6.4 Interrupt Interface

Function Name	Synopsis
<Pers>_isr	Process an interrupt generated by a device.
<Pers>_check_if_int_pending	Check if a device has generated an interrupt.
<Pers>_disable_ints	Disable interrupts from a device.
<Pers>_enable_ints	Re-enable interrupts from a device.

TABLE 15. Interrupt Interface Entry Points

Function:

<Pers>_isr

Prototype:

void <Pers>_isr(pers_dev_t *pC);

Synopsis

Handle an interrupt from the associated hardware device

Parameters

pC Pointer to the pers_dev_t structure associated with the interface on which interrupt occured.

Return Value

None

Detailed Description

When <Pers>_isr is called, the relevant bus module should have already determined that the associated device actually generated the interrupt, using <Pers>_check_if_int_pending, and performed any other OS specific process.

The only responsibility of <Pers>_isr is to handle and clear all pending interrupts on the hardware device.

Function:

<Pers>_check_if_int_pending

Prototype:

```
i_boolean_t <Pers>_check_if_int_pending(pers_dev_t *pC);
```

Synopsis

Determine if an interrupt is pending on the associated hardware device

Parameters

pC Pointer to the pers_dev_t structure associated with the interface to be checked.

Return Value

TRUE if an interrupt is pending, otherwise FALSE.

Detailed Description

Most operating systems, especially those supporting PCI devices, require a device driver to check if its associated hardware has generated an interrupt. This is because several devices might be sharing the same interrupt line. This function allows the ISR in the bus module to implement the appropriate behavior, checking if the hardware generated the interrupt, and returning the appropriate value to the OS, as well as calling the <Pers>_isr function if there is an interrupt pending. The following is a short piece of sample code showing how an bus module might use these functions to implement an ISR for an imaginary OS.

```
/* OS requires ISR to return TRUE if interrupt is pending, otherwise false */
/* ippci_isr() is registered with the OS as the interrupt handler */

i_boolean_t ippci_isr(bus_t *pbus)
{
    i_boolean_t status;
    status = (*pbus->ivector.check_if_int_pending)(pbus->child_handle);
    if(status) {
        (*pbus->ivector.isr_func)(pbus->child_handle)
    }
    return status;
}
```

Function:

<Pers>_disable_ints

Prototype:

```
void <Pers>_disable_ints(pers_dev_t *pC);
```

Synopsis

Disable all interrupts on the associated hardware.

Parameters

pC Pointer to the pers_dev_t structure associated with the interface whose interrupts should be disabled.

Return Value

None

Detailed Description

Some operating systems, including NT and Netware require device drivers to disable and enable interrupts on their associated hardware. This function allows the ISR in the bus module to provide the appropriate behavior.

Function:

<Pers>_enable_ints

Prototype:

```
void <Pers>_disable_ints(pers_dev_t *pC);
```

Synopsis

Enable interrupts on the associated hardware.

Parameters

pC Pointer to the pers_dev_t structure associated with the interface whose interrupts should be enable.

Return Value

None

Detailed Description

Some operating systems, including NT and Netware require device drivers to disable and enable interrupts on their associated hardware. This function allows the ISR in the bus module to provide the appropriate behavior.

6.5 Message Interface

6.5.1 Messages used by LAN personalities

Message Type	Synopsis
LAN_RUN_DIAGS_MSG	Run diagnostics for device.
LAN_RUN_DIAGS_RESP	
LAN_STACK_CON_MSG	Establish linkage to parent driver
LAN_STACK_CON_RESP	
LAN_STACK_DISCON_MSG	Remove linkage to parent driver
LAN_STACK_DISCON_RESP	
LAN_DEV_ENABLE_MSG	Enable device
LAN_DEV_ENABLE_RESP	
LAN_DEV_DISABLE_MSG	Disable device
LAN_DEV_DISABLE_RESP	
LAN_ADD_MCAST_MSG	Listen to the specified multicast address
LAN_ADD_MCAST_RESP	
LAN_DEL_MCAST_MSG	Stop listening to the specified multicast address
LAN_DEL_MCAST_RESP	
LAN_SET_PROMISC_MSG	Set promiscuous mode at the specified level
LAN_SET_PROMISC_RESP	
LAN_SET_DUPLEX_MSG	Set duplex mode as specified
LAN_SET_DUPLEX_RESP	
LAN_SET_MAC_MSG	Set MAC address to the specified address
LAN_SET_MAC_RESP	
LAN_SET_SPEED_MSG	Set LAN speed to the specified speed
LAN_SET_SPEED_RESP	

TABLE 16. LAN Personality Messages

Message Name

LAN_RUN_DIAGS_MSG

Synopsis

Run diagnostics for device.

Message Data

```
typedef struct {
     lan_type_t lan_dev_type;
     i_ubit32_t   diag_level;
} lan_run_diags_msg_t;
```

Response Name

LAN_RUN_DIAGS_RESP

Response Data

```
typedef struct {
     i_status_t status;
} lan_run_diags_resp_t;
```

Detailed description

Message Name

LAN_STACK_CON_MSG

Synopsis

Establish linkage to parent driver

Message Data

```
typedef struct {
     void *parent_handle;
     lan_type_t lan_dev_type; /* FDDI, 802.3, IP, etc */
     i_ubit32_t extra_tx_hdr_bytes;
     i_ubit32_t extra_rx_hdr_bytes;
     void (*receive_pkt)(void *parent_handle, packet_t *rcvd_pkt,
     addr_match_t addr_match);
     void (*tx_restart)(void *parent_handle);
     void (*tx_stop)(void *parent_handle);
} lan_stack_con_msg_t;
```

Response Name

LAN_STACK_CON_RESP

Response Data

```
typedef struct {
     i_status_t status;
     void *child_handle;
     bdm_t *pbdm;
     i_ubit8_t mac_addr[6];
     msg_status_t (*send_pkt)(void *child_handle, packet_t *tx_pkt);
} lan_stack_con_resp_t;
```

Detailed description

Message Name

LAN_STACK_DISCON_MSG

Synopsis

Remove linkage to parent driver

Message Data

```
typedef struct {
    i_ubit32_t no_info_required;
} lan_stack_discon_msg_t;
```

Response Name

LAN_STACK_DISCON_RESP

Response Data

```
typedef struct {
    i_status_t status;
} lan_stack_discon_resp_t;
```

Detailed description

Message Name

LAN_DEV_ENABLE_MSG

Synopsis

Enable device

Message Data

```
typedef struct {
     i_ubit32_t nodata;
} lan_enable_msg_t;
```

Response Name

LAN_DEV_ENABLE_RESP

Response Data

```
typedef struct {
     i_status_t status;
} lan_enable_resp_t;
```

Detailed description

Message Name

LAN_DEV_DISABLE_MSG

Synopsis

Listen to the specified multicast address

Message Data

```
typedef struct {
    i_ubit32_t no_info_required;
} lan_disable_msg_t;
```

Response Name

LAN_ADD_MCAST_RESP

Response Data

```
typedef struct {
    i_status_t status;
} lan_disable_resp_t;
```

Detailed description

Message Name

LAN_DEL_MCAST_MSG

Synopsis

Stop listening to the specified multicast address

Message Data

Response Name

LAN_DEL_MCAST_RESP

Response Data

Detailed description

Message Name

LAN_SET_PROMISC_MSG

Synopsis

Set promiscuous mode at the specified level

Message Data

Response Name

LAN_SET_PROMISC_RESP

Response Data

Detailed description

Message Name

LAN_SET_DUPLEX_MSG

Synopsis

Set duplex mode as specified

Message Data

Response Name

LAN_SET_DUPLEX_RESP

Response Data

Detailed description

Message Name

LAN_SET_MAC_MSG

Synopsis

Set MAC address to the specified address

Message Data

Response Name

LAN_SET_MAC_RESP

Response Data

Detailed description

Message Name

LAN_SET_SPEED_MSG

Synopsis

Set LAN speed to the specified speed

Message Data

Response Name

LAN_SET_SPEED_RESP

Response Data

Detailed description

6.5.2 LAN personality events

Events are sent asynchronously in response to a condition the sending module has detected. They have no expected response, and usually no associated data.

Message Type	Synopsis
LAN_LINK_UP_EVENT	Notification that a network link has been detected.
LAN_LINK_DOWN_EVENT	Notification that a link failure has occured.
LAN_HW_DEAD_EVENT	Notification that a hardware failure has been detected.

TABLE 17. LAN Personality Events

Message Name

LAN_LINK_UP_EVENT

Synopsis

Notification that a network link has been detected.

Message Data

None

Detailed description

This event informs any upper layer PVD's, along with the NIF that a physical network link has been detected. The PPD should attempt to detect or establish a network link during the "enable" phase, and once the link is established, it should send a LAN_LINK_UP_EVENT to its parent entity (PVD or NIF).

LAN_LINK_UP_EVENT should be sent only once after the link has been established. If the network link is lost, and LAN_LINK_DOWN_EVENT is sent, then when the link is re-established, LAN_LINK_UP_EVENT should be sent.

Message Name

LAN_LINK_DOWN_EVENT

Synopsis

Notification that a link failure has occured.

Message Data

None

Detailed description

This event informs any upper layer PVD's, along with the NIF that the physical network link has been lost. The PPD must continually monitor the status of its network link. When a link failure is detected, LAN_LINK_DOWN_EVENT should be sent to its parent entity (PVD or NIF).

LAN_LINK_DOWN_EVENT should not be sent without being preceded by a LAN_LINK_UP_EVENT. Specifically, upon intialization, LAN_LINK_DOWN_EVENT should not be sent until the link has been established, and a LAN_LINK_UP_EVENT has been sent. Once LAN_LINK_DOWN_EVENT has been sent, it should not be sent again, until the link is re-established, and a LAN_LINK_UP_EVENT has been sent.

Message Name

LAN_HW_DEAD_EVENT

Synopsis

Notification that a hardware failure has been detected.

Message Data

None

Detailed description

LAN_HW_DEAD_EVENT should be sent in the case where the PPD detects a hardware failure that cannot be recovered from at the local level. An example of such a failure might be a case where the hardware does not respond to a command in a given period of time.

The NIF will likely respond with a LAN_DEV_DISABLE_MSG, possibly followed by a LAN_DEV_ENABLE message to try to restore the hardware to a functioning state. Therefore, LAN_HW_DEAD_EVENT should only be used for serious failures. It should not be used for “permissable” failures such as underruns and overruns.

Although the hardware is placed in a “dead” state when a PPD receives a LAN_DEV_DISABLE_MSG, it should not send a LAN_HW_DEAD_EVENT in response. The NIF should assume that the LAN_DEV_DISABLE_RESP indicates the hardware is dead, in addition to acknowledging the previous message. This is to avoid a loop where the NIF sends LAN_DEV_DISABLE_MSG message, the PPD responds with LAN_HW_DEAD_EVENT, and then the NIF tries to “fix” the hardware with another LAN_DEV_DISABLE_MSG.

6.5.3 Messages used by SCSI personaliities

Message Type	Synopsis
SCSI_RUN_DIAGS_MSG	Run diagnostics for device.
SCSI_RUN_DIAGS_RESP	
SCSI_STACK_CON_MSG	Establish linkage to parent driver
SCSI_STACK_CON_RESP	
SCSI_STACK_DISCON_MSG	Remove linkage to parent driver
SCSI_STACK_DISCON_RESP	
SCSI_DEV_ENABLE_MSG	Enable device
SCSI_DEV_ENABLE_RESP	
SCSI_DEV_DISABLE_MSG	Disable device
SCSI_DEV_DISABLE_RESP	

TABLE 18. SCSI Personality Messages

7.0 Zone Interface Reference

7.1 OIM Interface

Function Name	Synopsis
oim_load_start	Call load functions for all Zone modules
oim_stop_unload	Call unload functions for all Zone modules
oim_add_pers	Add a personality to the list of known personalities
oim_remove_pers	Remove a personality from the list of known personalities
oim_add_bus	Add a bus to the list of known buses
oim_remove_bus	Remove a bus from the list of known buses
oim_add_card_instance	Add a personality instance to the list of known personality instances.
oim_remove_card_instance	Remove a personality instance from the list.
oim_ss_pvd_probe	Probe for virtual device instances for the given PVD in stable storage.
oim_ss_get_pvd_child	Look up the child of a given PVD instance in stable storage.
oim_ss_check_os_if	Check if a given PVD or PPD instance has an associated OS interface.

TABLE 19. OIM Function Interface

Function:

oim_load_start

Prototype:

```
i_status_t oim_load_start(void);
```

Synopsis

Call load functions for all Zone modules.

Parameters

Return Value

Detailed Description

Function:

oim_stop_unload

Prototype:

```
void oim_stop_unload(void);
```

Synopsis

Call unload functions for all Zone modules

Parameters

Return Value

Detailed Description

Function:

oim_add_pers

Prototype:

```
i_status_t oim_add_pers(pers_props_t *card_mod_props);
```

Synopsis

Add a personality to the list of known personalities

Parameters

Return Value

Detailed Description

Function:

oim_remove_pers

Prototype:

```
void oim_remove_pers(pers_props_t *card);
```

Synopsis

Remove a personality from the list of known personalities

Parameters

Return Value

Detailed Description

Function:

oim_add_bus

Prototype:

```
i_status_t oim_add_bus(bus_props_t *bus_props);
```

Synopsis

Add a bus to the list of known buses

Parameters

Return Value

Detailed Description

Function:

oim_remove_bus

Prototype:

```
void oim_remove_bus(bus_props_t *bus_props);
```

Synopsis

Remove a bus from the list of known buses

Parameters

Return Value

Detailed Description

Function:

oim_add_card_instance

Prototype:

```
oim_t *oim_add_pers_instance(pers_dev_t *Card);
```

Synopsis

Add a personality instance to the list of known personality instances.

Parameters

Return Value

Detailed Description

Function:

oim_remove_card_instance

Prototype:

```
void oim_remove_card_instance(oim_t *oim);
```

Synopsis

Remove a personality instance from the list.

Parameters

Return Value

Detailed Description

Function:

oim_ss_pvd_probe

Prototype:

```
snd_dev_id_t *oim_ss_pvd_probe(snd_dev_id_t *dev_id,
                               char *pvd_short_name,
                               i_ubit_32_t instance_num);
```

Synopsis

Probe for virtual device instances for the given PVD in stable storage.

Parameters

Return Value

Detailed Description

Function:

oim_ss_get_pvd_child

Prototype:

```
char *oim_ss_get_pvd_child(char *pvd_inst_name);
```

Synopsis

Look up the child of a given PVD instance in stable storage.

Parameters

Return Value

Detailed Description

Function:

oim_ss_check_os_if

Prototype:

```
i_boolean_t oim_ss_check_os_if(char *pvd_inst_name);
```

Synopsis

Check if a given PVD or PPD instance has an associated OS interface.

Parameters

Return Value

Detailed Description

7.2 NIF and SIF Interface

Function Name	Synopsis
nif_add_card_instance	Add a network personality instance to the NIF's list of known personality instances.
nif_remove_card_instance	Remove a network personality instance from the NIF's list of known personality instances.
sif_add_card_instance	Add a storagepersonality instance to the SIF's list of known personality instances.
sif_remove_card_instance	Remove a storage personality instance from the SIF's list of known personality instances.

TABLE 20. NIF and SIF Function Interface

Function:

nif_add_card_instance

Prototype:

```
void *nif_add_card_instance(pers_dev_t *Card, void *oim);
```

Synopsis

Add a network personality instance to the NIF’s list of known personality instances.

Parameters

Return Value

Detailed Description

Function:

nif_remove_card_instance

Prototype:

```
void nif_remove_card_instance(void *nif_ptr);
```

Synopsis

Remove a network personality instance from the NIF’s list of known personality instances.

Parameters

Return Value

Detailed Description

Function:

sif_add_card_instance

Prototype:

```
void *nif_add_card_instance(pers_dev_t *Card, void *oim);
```

Synopsis

Add a storage personality instance to the SIF’s list of known personality instances.

Parameters

Return Value

Detailed Description

Function:

sif_remove_card_instance

Prototype:

```
void nif_remove_card_instance(void *nif_ptr);
```

Synopsis

Remove a storage personality instance from the SIF’s list of known personality instances.

Parameters

Return Value

Detailed Description

7.3 OSM Interface

Previous versions of SND provided an OSM interface which was intended for use by all the other modules. These functions, except for osm_logmsg() are now obsolete and not documented here, although they are still available in existing Zone's for existing personalities. The OSM services shown below (except for osm_logmsg) are not intended to be called directly by arbitrary SND modules, and are provided for use by core modules to implement services which will be available to the other modules.

Function Name	Synopsis
osm_i_malloc	Allocate a block of memory of the specified size
osm_i_free	Free a block of memory previously allocated with osm_i_malloc()
osm_i_timeout	Schedule a timeout routine to be called after a specified interval.
osm_spinwait	Busy wait for a specified period of time.
osm_i_printstr	Print a string to an appropriate output device
osm_logmsg	Log a message to an appropriate logging area
osm_i_createlock	Allocate an OS spinlock
osm_i_freelock	Free an OS spinlock
OSM_I_LOCK	Acquire an OS spinlock
OSM_I_UNLOCK	Release an OS spinlock
osm_timestamp	Return a timestamp value which can be used for logging data.

Function:

osm_i_malloc

Prototype:

```
void *osm_i_malloc(i_ubit32_t size);
```

Synopsis

Allocate a block of memory of the specified size

Parameters

size	Number of bytes to allocate

Return Value

Returns a pointer to the memory range if the allocation was successful, otherwise returns NULL.

Detailed Description

Allocates *size* bytes of memory from the operating systems memory pool. This function is used by the i_mem to manage memory. It can be called directly if allocations must be performed before i_mem is initialized, but this should be avoided if at all possible.

Function:

osm_i_free

Prototype:

```
void osm_i_free(void *addr, i_ubit32_t size);
```

Synopsis

Free a block of memory previously allocated with osm_i_malloc()

Parameters

addr	Address of memory block previously allocated with osm_i_malloc().
size	Number of bytes to free. This must be the same value that was used in the call to osm_i_malloc().

Return Value

None

Detailed Description

osm_i_free() frees a block of memory previously allocated by osm_i_malloc(). Although the size must be specified, the size must be the same value which was used when the memory was allocated. For example, allocating 4k with osm_i_malloc(), then freeing 2k with osm_i_free() is forbidden. The size parameter exists to support operating systems whose internal memory free routines require a size parameter.

Function:

osm_i_timeout

Prototype:

```
void *osm_i_timeout(void (*trtn)(void *arg), void *arg,
                    i_ubit32_t tval_usecs);
```

Synopsis

Schedule a timeout routine to be called after a specified interval

Parameters

trtn	Pointer to a function to be called when the timeout period has expired.
arg	Argument to pass to the *trtn* function when the timeout period has expired.
tval_usecs	Number of microseconds to wait before calling the timeout function (*trtn*).

Return Value

A pointer to an OS specific structure corresponding to the timeout. If the timeout could not be scheduled, NULL is returned. The pointer is for use in a call to osm_i_untimeout() to cancel the timeout.

Detailed Description

Schedule a timeout period of *tval_usecs*, after which the function pointed to by *trtn* will be called. This function is intended for use by the i_timer code, and should not be called directly by other modules.

Function:

osm_spinwait

Prototype:

```
void osm_spinwait(i_ubit32_t usecs_to_wait);
```

Synopsis

Busy wait for a specified period of time.

Parameters

usecs_to_wait Number of microseconds to busywait.

Return Value

None

Detailed Description

Function:

osm_i_printstr

Prototype:

```
void osm_i_printstr(char *outstring);
```

Synopsis

Print a string to an appropriate output device

Parameters

outstring Pointer to the beginning of a character string to be printed.

Return Value

None

Detailed Description

Prints the string pointed to by outstring to an appropriate output device, such as the console device (in most unices) or the kernel debugger (for Windows NT). This function performs no formatting, and is intended for use by the i_string functions.

Function:

osm_logmsg

Prototype:

```
void osm_logmsg(i_ubit32_t class, i_ubit32_t module_id,
                i_ubit32_t msg_id, char *devname,
                char *fmt, ...);
```

Synopsis

Log a message to an appropriate logging area

Parameters

class Class of message to be logged. Must be one of the following values:

OSM_INFO

Message is for information only, and no action is required.

OSM_WARN

Something has happened which may result in problems later, but there is no immediate problem

OSM_ERROR

An error has occured, resulting is a dropped frame, aborted SCSI command etc. Recovery is possible without user intervention.

OSM_FATAL

An error has occured, and either no recovery is possible, or user intervention is required for recover (in which case the required intervention should be described within the message).

OSM_PANIC

An unrecoverable condition was encountered, which compromises system integrity. Zones should call some sort of “panic” or “bugcheck” function if a message of this type is logged, which will halt the system.

module_id Module ID of message to be logged

msg_id Message number for the logged message.

devname SND open firmware string for the device the logged message relates to. If there is no related device, then NULL may be passed.

fmt Format string for the logged message, mostly equivelant to printf().

... Zero or more parameters for use in the format string.

Return Value

None

Detailed Description

osm_logmsg() is intended to map onto OS level logging facilities if available, or alternatively to format messages in a standardized way and print them to an appropriate output device. Every message must have a unique message ID, and every module which logs messages must have a unique log ID. For OS's which do not have built in logging facilities, the following output format is suggested:

<dev_name>:<class>:<module_id>:<msg_id>:<fmt>

Function:

osm_i_createlock

Prototype:

```
void *osm_i_createlock(bus_t *pBUS, i_boolean_t prevent_intr);
```

Synopsis

Allocate an OS spinlock

Parameters

pBUS Pointer to a bus_t structure, which indicates an interrupt level which the lock must synchronize against. Must be specified if *prevent_intr* is TRUE. Can be NULL if *prevent_intr* is FALSE.

prevent_intr Flag to indicate whether the interrupt level should be raised to prevent device interrupts before acquiring the lock. If TRUE, then device interrupts will be suppressed while the lock is held. If FALSE, then device interrupts are allowed while the lock is held.

Return Value

Pointer to an OS specific lock structure.

Detailed Description

Allocates and initializes an OS specific lock structure, configured to synchronize with interrupt level if prevent_intr is TRUE. This function is intended to be called by the i_lock module.

Function:

osm_i_freelock

Prototype:

```
void osm_i_freelock(void *os_lock);
```

Synopsis

Free an OS spinlock

Parameters

os_lock Pointer to a lock previously allocated with osm_i_createlock.

Return Value

None

Detailed Description

Function:

OSM_I_LOCK

Prototype:

```
void OSM_I_LOCK(void *os_lock_struct);
```

Synopsis

Macro which acquires an OS spinlock

Parameters

os_lock_struct Pointer to an OS specific lock structure previously created by a call to osm_i_createlock().

Return Value

none

Detailed Description

Acquires a spinlock previous created with osm_i_createlock(). Should be implemented as a macro for improved performance. Intended to be used by the i_lock module.

Function:

OSM_I_UNLOCK

Prototype:

```
void OSM_I_UNLOCK(void *os_lock_struct);
```

Synopsis

Release an OS spinlock

Parameters

os_lock_struct Pointer to an OS specific lock structure previously created by a call to osm_i_createlock().

Return Value

None

Detailed Description

Release a spinlock previous acquired with OSM_I_LOCK. Should be implemented as a macro for improved performance. Intended to be used by the i_lock module.

Function:

osm_timestamp

Prototype:

```
i_ubit32_t osm_timestamp();
```

Synopsis

Return a timestamp value which can be used for logging data.

Parameters

None

Return Value

32 bit timestamp, in units which are OS specific.

Detailed Description

This service is intended to provide timestamps for logging and performance tuning purposes. Some environments might not be able to implement this service at all (the stub must still be provided) so code should never require this service to be functioning for correct operation.

7.4 Bus Interface

Bus services may be called with or without a personality region lock.

Function Name	Synopsis
bus_dma_limits_t	Structure describing DMA capabilities of hardware.
bus_dma_init	Prepare bus module to handle DMA operations (mapping, etc).
bus_dma_done	Converse of bus_dma_init(). Free any resources allocated during bus_dma_init().
bus_dma_handle_alloc	Allocate a DMA handle for a DMA transfer across the specified bus. **Cannot be called from personality region**.
bus_dma_handle_free	Free a DMA handle previously allocated with bus_dma_handle_alloc. **Cannot be called from personality region**.
bus_dma_map	Map a region of memory for DMA across the specified bus. **Cannot be called from personality region**.
bus_dma_unmap	Unmap a region of memory previously mapped with bus_dma_map. **Cannot be called from personality region.**
bus_query_max_mm_len	Determine the maximum length of contiguous physical memory which can be allocated using bus_mm_alloc
bus_mm_alloc	Allocate a region of memory, and map it to the specified bus.
bus_mm_zalloc	Allocate and zero a region of memory, and map it to the specified bus.
bus_mm_free	Free a region of memory previously allocated with bus_mm_alloc or bus_mm_zalloc.
bus_mm_sync	Synchronize the cache with the main memory within a region of memory allocated with bus_mm_alloc or bus_mm_zalloc.
bus_cache_sync	Synchronize the cache with the main memory at a given location.
bus_isr_add	Register an ISR for a device on the specified bus.
bus_isr_remove	De-register an ISR previously registered with bus_isr_add.

Structure Name:

bus_dma_limits_t

Definition:

TBD

Synopsis

Field Descriptions

Function:

bus_dma_init

Prototype:

```
i_status_t bus_dma_init(bus_t *bus, bus_dma_limits_t *pBDL);
```

Synopsis

Prepare bus module to handle DMA operations (mapping, etc).

Parameters

Return Value

Detailed Description

Function:

bus_dma_done

Prototype:

```
void i_status_t bus_dma_done(bus_t *bus);
```

Synopsis

Converse of bus_dma_init(). Free any resources allocated during bus_dma_init().

Parameters

Return Value

Detailed Description

Function:

bus_dma_handle_alloc

Prototype:

```
dma_handle_t *bus_dma_handle_alloc(bus_t *pbus,
                                   bus_data_direction_t direction,
                                   i_ubit32_t sglistbytes);
```

Synopsis

Allocate a DMA handle for a DMA transfer across the specified bus.

Parameters

Return Value

Detailed Description

Function:

bus_dma_handle_free

Prototype:

```
void bus_dma_handle_free(dma_handle_t *dma_handle);
```

Synopsis

Free a DMA handle previously allocated with bus_dma_handle_alloc.

Parameters

Return Value

Detailed Description

Function:

bus_dma_map

Prototype:

```
i_status_t bus_dma_map(bus_t *pBUS, dma_handle_t *dma_handle);
```

Synopsis

Map a region of memory for DMA across the specified bus

Parameters

Return Value

Detailed Description

Function:

bus_dma_unmap

Prototype:

```
void bus_dma_unmap(bus_t *pBUS, dma_handle_t *dma_handle);
```

Synopsis

Unmap a region of memory previously mapped with bus_dma_map.

Parameters

Return Value

Detailed Description

Function:

bus_query_max_mm_len(bus_t *pbus);

Prototype:

```
i_ubit32_t *bus_query_max_mm_len(bus_t *pbus);
```

Synopsis

Determine the maximum length of contiguous physical memory which can be allocated using bus_mm_alloc.

Parameters

pbus Pointer to a bus_t structure describing the bus which the physical memory will be mapped against.

Return Value

The number of bytes of contiguous memory which can be allocated using bus_mm_alloc. This result will always be a power of two.

Detailed Description

In most systems, this service returns the page size, since a physical page is always contiguous. Certain systems with “mapping registers” may be able to create areas of memory which are contiguous from the perspective of a device on the bus, yet are larger than a physical page. This facility is sometimes refered to as “I/O Virtual Memory”

Function:

bus_mm_alloc

Prototype:

```
immh_t *bus_mm_alloc(bus_t *pbus, i_ubit32_t size,
                     i_ubit32_t alignment,
                     i_ubit32_t dmah_flags);
```

Synopsis

Allocate a region of memory, and map it to the specified bus.

Parameters

Return Value

Detailed Description

Function:

bus_mm_zalloc

Prototype:

```
immh_t *bus_mm_zalloc(bus_t *pbus, i_ubit32_t size,
                      i_ubit32_t alignment,
                      i_ubit32_t dmah_flags);
```

Synopsis

Allocate and zero a region of memory, and map it to the specified bus.

Parameters

Return Value

Detailed Description

Function:

bus_mm_free

Prototype:

```
void bus_mm_free(immh_t *mmh);
```

Synopsis

Free a region of memory previously allocated with bus_mm_alloc or bus_mm_zalloc.

Parameters

Return Value

Detailed Description

Function:

bus_mm_sync

Prototype:

```
void bus_mm_sync(immh_t *mmh,
                 void *addr,
                 i_ubit32_t length
                 BUS_Data_Direction direction);
```

Synopsis

Synchronize the cache with the main memory within a region of memory allocated with bus_mm_alloc or bus_mm_zalloc.

Parameters

Return Value

Detailed Description

Function:

bus_cache_sync

Prototype:

```
void bus_cache_sync(bus_t *bus, dma_handle_t *dmah,
                    void *addr, i_ubit32_t size,
                    BUS_Data_Direction direction, void (*callback)(),
                    void *arg1, void *arg1);
```

Synopsis

Synchronize the cache with the main memory at a given location.

Parameters

Return Value

Detailed Description

Function:

bus_isr_add

Prototype:

```
i_status_t bus_isr_add(bus_t *bus, bus_ivector_t *ivector,
                       pers_dev_t *isr_arg);
```

Synopsis

Register an ISR for a device on the specified bus.

Parameters

Return Value

Detailed Description

Function:

bus_isr_remove

Prototype:

```
void bus_isr_remove(bus_t *bus);
```

Synopsis

De-register an ISR previously registered with bus_isr_add.

Parameters

Return Value

Detailed Description

7.5 PIO Interface

PIO operations make use of the bus module. All operations currently defined are actually macros which call function pointers contained within the pio handle or bus structure.

Function Name	Synopsis
pio_map	Map an area of adapter I/O space to host memory for access with PIO operations. Also can be used to get a handle to access host memory with PIO operations with correct byte ordering.
pio_unmap	Release a pio handle previously allocated with pio_map, along with any associated resources.
pio_in_ubit8	Read an 8 bit value from a specified offset within a pio mapped region.
pio_in_ubit16	Read a 16 bit value from a specified offset within a pio mapped region.
pio_in_ubit32	Read a 32 bit value from a specified offset within a pio mapped region.
pio_in	Read one or more 8 bit values starting from a specified offset within a pio mapped region into a buffer.
pio_out_ubit8	Write an 8 bit value to a specified offset within a pio mapped region.
pio_out_ubit16	Write a 16 bit value to a specified offset within a pio mapped region.
pio_out_ubit32	Write a 32 bit value to a specified offset within a pio mapped region.
pio_out	Write one or more 8 bit values starting at specified offset within a pio mapped region from a buffer.
pio_protect_region_from_dma	Disable DMA operations on a particular bus. Optional, but required for 4824.
pio_allow_dma_to_region	Re-enable DMA operations on a particular bus. Optional, but required for 4824.
pio_sync_region	Synchronize cache within a PIO mapped region of type PIO_DMA_MEM.
pio_sync_mm	Synchronize cache within an mapped memory handle (i_mmh_t).

Function:

pio_map

Prototype:

```
pio_t *pio_map(bus_t *bus,
               pio_space_t *space_type,
               i_ubit32_t base_addr,
               i_ubit32_t offset,
               i_ubit32_t len);
```

Synopsis

Map an area of adapter I/O space to host memory for access with PIO operations. Also can be used to get a handle to access host memory with PIO operations with correct byte ordering.

Parameters

Return Value

Detailed Description

Function:

pio_unmap

Prototype:

```
void pio_unmap(pio_t *pioh);
```

Synopsis

Release a pio handle previously allocated with pio_map, along with any associated resources.

Parameters

Return Value

Detailed Description

Function:

pio_in_ubit8

Prototype:

```
i_ubit8_t pio_in_ubit8(pio_t *pioh, i_ubit32_t offset);
```

Synopsis

Read an 8 bit value from a specified offset within a pio mapped region.

Parameters

Return Value

Detailed Description

Function:

pio_in_ubit16

Prototype:

```
i_ubit16_t pio_in_ubit16(pio_t *pioh, i_ubit32_t offset);
```

Synopsis

Read a 16 bit value from a specified offset within a pio mapped region.

Parameters

Return Value

Detailed Description

Function:

pio_in_ubit32

Prototype:

```
i_ubit32_t pio_in_ubit32(pio_t *pioh, i_ubit32_t offset);
```

Synopsis

Read a 32 bit value from a specified offset within a pio mapped region.

Parameters

Return Value

Detailed Description

Function:

pio_in

Prototype:

```
void pio_in(pio_t *pioh, i_ubit32_t offset,
            i_ubit32_t length, void *dst_buffer);
```

Synopsis

Read one or more 8 bit values starting from a specified offset within a pio mapped region into a buffer.

Parameters

Return Value

Detailed Description

Function:

pio_out_ubit8

Prototype:

```
void pio_out_ubit8(pio_t *pioh, i_ubit32_t offset, i_ubit8_t data);
```

Synopsis

Write an 8 bit value to a specified offset within a pio mapped region.

Parameters

Return Value

Detailed Description

Function:

pio_out_ubit16

Prototype:

```
void pio_out_ubit16(pio_t *pioh, i_ubit32_t offset, i_ubit16_t data);
```

Synopsis

Write a 16 bit value to a specified offset within a pio mapped region.

Parameters

Return Value

Detailed Description

Function:

pio_out_ubit32

Prototype:

```
void pio_out_ubit32(pio_t *pioh, i_ubit32_t offset, i_ubit32_t data);
```

Synopsis

Write a 32 bit value to a specified offset within a pio mapped region.

Parameters

Return Value

Detailed Description

Function:

pio_out

Prototype:

```
void pio_out(pio_t *pioh, i_ubit32_t offset,
            i_ubit32_t length, void *src_buffer);
```

Synopsis

Write one or more 8 bit values starting at specified offset within a pio mapped region from a buffer.

Parameters

Return Value

Detailed Description

Function:

pio_protect_region_from_dma

Prototype:

```
void pio_protect_region_from_dma(pio_t *pioh);
```

Synopsis

Disable DMA operations on a particular bus. Optional, but required for 4824.

Parameters

Return Value

Detailed Description

Function:

pio_allow_dma_to_region

Prototype:

```
void pio_allow_dma_to_region(pio_t *pioh);
```

Synopsis

Re-enable DMA operations on a particular bus. Optional, but required for 4824.

Parameters

Return Value

Detailed Description

Function:

pio_sync_region

Prototype:

```
void pio_sync_region(pio_t *pioh, i_ubit32_t offset, i_ubit32_t length
                     bus_cache_op_t cache_op);
```

Synopsis

Synchronize cache within a PIO mapped region of type PIO_DMA_MEM.

Parameters

Return Value

Detailed Description

Function:

pio_sync_mm

Prototype:

```
void pio_sync_mm(i_mmh_t *mmh, i_ubit32_t offset, i_ubit32_t length,
                 bus_cache_op_t cache_op);
```

Synopsis

Synchronize cache within an mapped memory handle (i_mmh_t).

Parameters

Return Value

Detailed Description

7.6 BDM Interface

Structure Name	Synopsis
scsi_cb_t	Structure corresponding to a SCSI command
net_cb_t	Structure corresponding to a network packet
bdm_buf_t	Structure defining a region of mapped memory, generally associated with another structure such as scsi_cb_t or net_cb_t.

TABLE 21. BDM Interface Structures

Structure Name

scsi_cb_t

Synopsis

Structure corresponding to a SCSI command.

Definition

```
typedef struct scsi_cb_s {
      struct scsi_cb_s *next;
      struct scsi_cb_s *prev;
      struct bdm_s *pbdm;
      bus_data_direction_t direction;
      i_ubit8_t *cdb;
      i_ubit32_t cdb_len;
      target_id_t target_id;
      lun_id_t lun;
      i_ubit32_t timeout_msecs;
      i_ubit32_t retries;
      i_ubit8_t *sense_data;
      i_ubit32_t sense_data_len;
      i_ubit32_t residual_count;
      i_ubit32_t flags;
      bdm_buf_t *data;
      /* <Private Data> */
} scsi_cb_t;
```

Description of Fields

next	Used by queuing functions for adding and removing from linked lists
prev	Used by queuing functions for adding and removing from linked lists
pbdm	Pointer to the bdm_t structure associated with the interface which the SCSI command is intended for.
norm_completion	Function pointer which should be called by the SCSI PPD or PVD after the command has been completed successfully.
error_completion	Function pointer which should be called by the SCSI PPD or PVD if the command cannot be completed successfully.
cdb	Pointer to the cdb for this SCSI command
cdb_len	Lengh of the cdb for this SCSI command
target_id	Destination SCSI Target ID for this command
lun	Destitation LUN for this command

timeout_msecs Number of milliseconds to wait for the command to complete before indicating failure. A value of ‘0’ indicates the command should never timeout (and probably indicates the OS or SIF is handling timeouts internally).

retries Number to times to resend a command if it fails initially

sense_data Sense data for this command. Filled in by FCP after command has been processed by hardware.

sense_data_len Length of sense data for this command. Filled in by FCP after command has been processed by hardware.

residual_count Bytes remaining for this command. Filled in by FCP after command has been processed by hardware.

flags No flags are currently defined - but need to used this field to hold status flags which would be filled in by FCP after command has been processed by hardware.

data Pointer to one or more bdm_buf_t structures which describe the data buffer associated with the SCSI command.

<Private Data>

Additional Comments

Structure Name

net_cb_t

Synopsis

Structure corresponding to a network packet. Only used for transmits

Definition

```
typdef struct net_cb_s {
      struct net_cb_s *next;
      struct bdm_s *pbdm;
      void (*norm_completion)(struct net_cb_s *buf);
      void (*error_completion)(struct net_cb_s *buf,
                               bdm_error_t error);
      bdm_buf_t *data;
      /* <Private Data> */
} net_cb_t;
```

Description of Fields

next	Used by queuing functions for adding and removing from linked lists.
pbdm	Pointer to the bdm_t structure associated with the interface which the packet is intended for.
norm_completion	Function pointer which should be called by the network PPD or PVD after the packet has been sent successfully.
error_completion	Function pointer which should be called by the network PPD or PVD if the packet could not be sent successfully.
data	Pointer to one or more bdm_buf_t structures which describe the data buffer associated with the network packet.

Additional Comments

Structure Name

bdm_buf_t

Synopsis

Structure defining a region of mapped memory, generally associated with another structure such as scsi_cb_t or net_cb_t.

Definition

```
typedef struct bdm_buf_s {
      struct bdm_buf_s *next; /* Used for queuing on SLL */
      struct bdm_buf_s *chain;/* Used for chaining together */
      struct bdm_s *pbdm; /* Pointer to source BDM */
      bus_data_direction_t data_direction;
      i_ubit8_t *virt_data_ptr;
      i_ubit32_t virt_data_len;
      i_ubit32_t total_data_len;
      i_ubit32_t total_psg_valid;
      i_ubit32_t psglist_valid;
      i_psglist_t *psglist;
      /* <Private Data> */
```

} bdm_buf_t;

Description of Fields

next For use be queuing functions to add or remove a bdm_buf_t from a queue of bdm_buf_t structures.

chain For chaining bdm_buf_t structures together to allow larger areas of memory. For example, a scsi_cb_t structure or net_cb_t structure points to one or more bdm_buf_t structures linked together which describe the data buffers for the operation.

pbdm Pointer to the BDM structure which was used to allocate this buffer.

virt_data_ptr This field is implementation specific unless the following conditions are satisfied:

- The bdm_buf_t is the first (or only) on the chain of bdm_buf_t structures.
- BDL_txhdr_size (for transmit buffers) or BDL_rxhdrsize (for receive buffers) is non-zero.

If these conditions are satisified, virt_data_ptr will point to a virtually contiguous area of memory at least BDM_txhdrsize or BDM_rxhdrsize (for transmit or receive buffers respectively).

total_data_len If the bdm_buf_t is the first (or only) on the chain of bdm_buf_structure, this field contains the total length of the data describe by all the psglists on all the bdm_buf_t structures on the chain. If the bdm_buf_t is not the first, total_data_len is undefined.

total_psg_valid If the bdm_buf_t is the first (or only) on a chain of bdm_buf_t structures, this field contains the total number of physical scatter/gather elements in all the psglists in all the bdm_buf_t structures on the chain. If the bdm_buf_t is not the first on the chain, total_psg_valid is undefined.

psglist_valid Indicates the number of valid entries in the psglist for this bdm_buf_t.

psglist Scatter/Gather list of physical addresses and lengths for the memory described by this bdm_buf_t.

<Private Data>

Additional Comments

May be called while holding a personality region lock.

Function Name	Synopsis
bdm_connect	Get BDM handle allowing use of BDM services
bdm_disconnect	Release BDM handle
bdm_tx_buf_alloc	Allocate bdm_buf_t for outbound DMA
bdm_buf_sync	Synchronize cached buffer contents with main memory, and map buffer if it has not already been mapped. Final operation on bdm_buf_t prior to outbound DMA or first operation after inbound DMA before accessing the buffer contents.
bdm_tx_buf_free	Free bdm_buf_t previously allocated with bdm_tx_buf_alloc. Only used for networking.
bdm_rx_buf_alloc	Allocate bdm_buf_t for inbound DMA
bdm_rx_buf_peek	Copied a specified number of bytes from the beginning of the buffer into a character array.
bdm_rx_buf_free	Free inbound DMA buffer to BDM.
bdm_del_buf_hdr	Delete a specified number of bytes from the beginning of the buffer
bdm_xlate_<OS_NET_BUF>	(optional zone function) Translate an OS representation of a network packet into a net_cb_t structure.
bdm_xlate_<OS_SCSI_BUF>	(optional zone function) Translate an OS representation of a SCSI request into a scsi_cb_t structure.
bdm_buf_to_<OS_NET_BUF>	(optional zone function) Translate a bdm_buf_t structure into the OS representation of a network packet.

TABLE 22. BDM Interface Functions

Function:

bdm_connect

Prototype:

```
bdm_t *bdm_connect(pers_dev_t *pC,
                   bus_dma_limits_t *pbdma);
```

Synopsis

Get BDM handle allowing use of BDM services.

Parameters

pC

pbdma

Return Value

Detailed Description

All modules related to a particular logical device must use the same bdm structure. If a PVD has two PPD child devices, it will stack connect to both of them, causing them each to get a bdm_t structure. The PVD should then sent a message (currently undefined) to one of the PVD's, telling it to free its bdm_t and use the bdm_t from the other PPD. This is to prevent freeing resources allocated from one bdm_t into another bdm_t.

A PPD which is bound to multiple PVD's may call bdm_connect multiple times. Again, the rule is one bdm_t per logical adapter.

Function:

bdm_disconnect

Prototype:

```
void bdm_disconnect(bdm_t *pbdm);
```

Synopsis

Release BDM handle.

Parameters

pbdm

Return Value

Detailed Description

Function:

bdm_tx_buf_alloc

Prototype:

```
bdm_buf_t *bdm_tx_buf_alloc(bdm_t *pbdm, i_ubit32_t buf_size);
```

Synopsis

Allocate mapped bdm_buf_t for outbound DMA

Parameters

pbdm

buf_size

Return Value

Detailed Description

Function:

bdm_buf_sync

Prototype:

```
void bdm_buf_sync(bdm_buf_t *prep_buf, void *cb_arg1, void *cb_arg2,
                  void (*sync_cb)(bdm_buf_t *buf, void *cb_arg1,
                  void *cb_arg2, i_status_t status));
```

Synopsis

Synchronize cached buffer contents with main memory, and map buffer if it has not already been mapped. Final operation on bdm_buf_t prior to outbound DMA or first operation after inbound DMA before accessing the buffer contents.

Parameters

prep_buf	The buffer to be synchronized with cache. For inbound DMA buffers, prep_buf->total_data_len should be updated to reflect the length of the received data
cb_arg1	First argument passed to callback function
cb_arg2	Second argument passed to callback function
sync_cb	Callback function -- this function will be called with the buffer has been synchronized with cache.

Return Value

None

Detailed Description

Only PPD’s should call bdm_buf_sync.

Function:

bdm_tx_buf_free

Prototype:

```
void bdm_tx_buf_free(bdm_buf_t *bufp);
```

Synopsis

Free bdm_buf_t previously allocated with bdm_tx_buf_alloc.

Parameters

bufp

Return Value

Detailed Description

Function:

bdm_rx_buf_alloc

Prototype:

```
bdm_buf_t *bdm_rx_buf_alloc(bdm_t *bdm, i_ubit32_t size);
```

Synopsis

Allocate bdm_buf_t for inbound DMA

Parameters

pbdm

size

Return Value

Detailed Description

Function:

bdm_rx_buf_peek

Prototype:

```
void bdm_rx_buf_peek(bdm_buf_t *rx_buf,
                     void *hdr_buf, i_ubit32_t hdr_len);
```

Synopsis

Copied a specified number of bytes from the beginning of the buffer into a character array.

Parameters

rx_buf

hdr_buf

hdr_len

Return Value

Detailed Description

Function:

bdm_rx_buf_free

Prototype:

```
void bdm_rx_buf_free(bdm_buf_t *rx_buf);
```

Synopsis

Free inbound DMA buffer to BDM.

Parameters

rx_buf

Return Value

None

Detailed Description

Function:

bdm_del_buf_hdr

Prototype:

```
void bdm_del_buf_hdr(bdm_buf_t *buf, i_ubit32_t hdr_len);
```

Synopsis

Delete a specified number of bytes from the beginning of the buffer

Parameters

buf

hdr_len

Return Value

None

Detailed Description

Function:

bdm_xlate_<OS_NET_BUF>

Prototype:

```
net_cb_t *bdm_xlate_<OS_NET_BUF>(bdm_t *bdm, <OS_NET_BUF> *os_buf);
```

Synopsis

(optional zone function) Translate an OS representation of a network packet into a net_cb_t structure.

Parameters

bdm

os_buf

Return Value

Detailed Description

Function:

bdm_xlate_<OS_SCSI_BUF>

Prototype:

```
scsi_cb_t *bdm_xlate_<OS_SCSI_BUF>(bdm_t *bdm, <OS_SCSI_BUF> *os_buf);
```

Synopsis

(optional zone function) Translate an OS representation of a SCSI command into a bdm_buf_t structure.

Parameters

bdm

os_buf

Return Value

Detailed Description

Function:

bdm_buf_to_<OS_NET_BUF>

Prototype:

```
<OS_NET_BUF> *bdm_buf_to_<OS_NET_BUF>(net_cb_t *bdm,i_ubit32_t gflags);
```

Synopsis

(optional zone function) Translate a bdm_buf_t structure into the OS representation of a network packet.

Parameters

bdm

gflags

Return Value

Detailed Description

8.0 Bridge Interface Reference

TBD

9.0 Using native code in personality modules

TBD

10.0 SND 2.x -> SND 3.x Delta

TBD