SNF™
Application Programming Interface

Version 3.0.13.50838

January 5, 2018

Contents

Chapter 1

SNF API Reference

1.1 Sniffer Documentation

The Sniffer User Guide covers NIC and software installation as well as instructions on how to use Sniffer with libpcap-based applications. It is available from https://cspi.force.com/customersupport/

1.2 SNF API Reference

The SNF API is available for applications that require tighter integration than libpcap with Sniffer. When used in its simplest form, the library resembles Libpcap in that the implementation expects a single thread to make successive calls to a receive function (snf_ring_recv) to obtain the next available packet. Under a more advanced form, Sniffer implements a variation of the Receive-Side Scaling (RSS) feature that is present in some 10-Gigabit Ethernet drivers. However, Sniffer takes the additional step of implementing the RSS feature as multiple user-level zero-copy receive rings. Making the rings available in userspace provides two important advantages over all existing kernel-based packet capture solutions:

- No system calls are required to retrieve packets from a kernel-level queue, and users have zero-copy access to incoming packets;
- There is no need to funnel all incoming packets through a single Libpcap-like device as would be the case in kernel-based methods, even if RSS rings are allocated in the kernel.

1.2.1 SNF API with Receive-Side Scaling

Sniffer translates the RSS feature into multiple rings in that data is hashed across many receive rings (or buffers or slices as is referenced in the myri10ge documentation). This feature assumes that users maintain a 1-to-1 relationship between user threads and rings. With each new call to snf_ring_recv, it is assumed that the previous packet in the ring has been completely consumed.

By default, the Sniffer implementation uses a deterministic hashing function to make sure that packets that are contained in a particular TCP or UDP flow are always delivered to the same ring (and hence to the same analysis thread). This hashing function resembles the hashing mechanisms used in existing RSS drivers.

1.2.2 SNF API with Duplication

While multiple rings are primarily designed to partition the incoming packet capture across multiple capture consuming rings, it is also possible to force each received packet to be duplicated into each ring such that every consuming ring obtains its own copy of every incoming packet. The duplication is handled by the Sniffer software on the host where there is typically plenty of memory bandwidth compared to the PCIe bus. Packet duplication can be enabled by setting the SNF_F_RX_DUPLICATE flag in snf_open.

1.2.3 SNF API with Port Aggregation (Merging)

With Sniffer 2.0, it is now possible to logically aggregate packets from two or more Ethernet ports. The functionality can be extended through to consumers that employ RSS or duplication. This feature can be enabled in snf_open by setting the SNF_F_AGGREGATE_PORTMASK flag and passing a bitmask of ports to aggregate in the **portnum** parameter. As a convenience, functions are also available to return portmasks for valid ports (snf_getportmask_valid) and active ports (snf_getportmask_linkup). As a result of calling snf_ring_recv, packets from one or more ports will be received.

1.3 Examples

Tests are available from **bin/tests** of the install directory in binary form and in **share/doc/examples** in source form. These tests mostly show different aspects of the SNF API and how to use its features.

snf_simple_recv.c: Simplest example of how to receive packets

snf_multi_recv.c: How to receive packets with multiple rings

snf_bridge.c: Example of how to use SNF to create a transparent bridge to analyze traffic on one device and replay it on another

snf_pktgen.c: How to generate packets for injection

snf_replay.c: Example that uses SNF-level injection to replay a .pcap capture file

snf_basic_diags: Basic internal diagnostics which can be useful to verify that everything works as expected (source code not available)

Chapter 2

Module Index

2.1 API Reference

Here is a list of all modules:

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

Module Documentation

5.1 SNF API Reference

API Reference for SNF and usage model.

Data Structures

- struct snf_ifaddrs
- struct snf_recv_req
- struct snf_ring_qinfo
- struct snf_ring_portinfo
- struct snf_ring_stats

Modules

- Receive-Side Scaling (RSS)
- Open flags for process-sharing, port aggregation and packet duplication
- Packet injection
- Packet reflect to netdev (kernel stack)

Defines

- #define SNF_VERSION_API 8

Typedefs

- typedef struct snf_handle * snf_handle_t

  opaque snf device handle
- typedef struct snf_ring * snf_ring_t

  opaque snf ring handle

Enumerations

- enum snf_link_state { **SNF_LINK_DOWN** = 0, **SNF_LINK_UP** = 1 }
- enum snf_timesource_state {
  SNF_TIMESOURCE_LOCAL = 0, SNF_TIMESOURCE_EXT_UNSYNCED, SNF_TIMESOURCE_EXT_-
  SYNCED, SNF_TIMESOURCE_EXT_FAILED,
  SNF_TIMESOURCE_ARISTA_ACTIVE, SNF_TIMESOURCE_PPS }

Functions

- int snf_init (uint16_t api_version)

  Initialize Sniffer Library with api_version == SNF_VERSION_API.
- int snf_set_app_id (int32_t id)

  Set the application ID.
- int snf_getifaddrs (struct snf_ifaddrs **ifaddrs_o)
- void snf_freeifaddrs (struct snf_ifaddrs *ifaddrs)
- int snf_getportmask_valid (uint32_t *mask_o, int *cnt_o)
- int snf_getportmask_linkup (uint32_t *mask_o, int *cnt_o)
- int snf_open (uint32_t portnum, int num_rings, const struct snf_rss_params *rss_params, int64_t dataring_sz, int flags, snf_handle_t *devhandle)

  Open device for single or multi-ring operation.
- int snf_open_defaults (uint32_t portnum, snf_handle_t *devhandle)

  Open device for single or multi-ring operation.
- int snf_start (snf_handle_t devhandle)
- int snf_stop (snf_handle_t devhandle)
- int snf_get_link_state (snf_handle_t devhandle, enum snf_link_state *state)
- int snf_get_timesource_state (snf_handle_t devhandle, enum snf_timesource_state *state)
- int snf_get_link_speed (snf_handle_t devhandle, uint64_t *speed)
- int snf_close (snf_handle_t devhandle)

  Close port.
- int snf_ring_open (snf_handle_t devhandle, snf_ring_t *ringh)
- int snf_ring_open_id (snf_handle_t devhandle, int ring_id, snf_ring_t *ringh)
- int snf_ring_close (snf_ring_t ringh)
- int snf_ring_recv (snf_ring_t ringh, int timeout_ms, struct snf_recv_req *recv_req)

  Receive next packet from a receive ring.
- int snf_ring_portinfo_count (snf_ring_t ring, int *count)

  For aggregated rings, return the number of physical subrings. If the ring is not aggregated, the count is set to 1.
- int snf_ring_portinfo (snf_ring_t ring, struct snf_ring_portinfo *portinfo)

  Returns information for the ring. For aggregated rings, returns information for each of the physical rings. It is up to the user to make sure they have allocated enough memory to hold the information for all the physical rings in an aggregated ring.
- int snf_ring_recv_qinfo (snf_ring_t ring, struct snf_ring_qinfo *)

  Return queue information from ring.
- int snf_ring_recv_many (snf_ring_t ring, int timeout_ms, struct snf_recv_req *req_vector, int nreq_in, int *nreq-
  _out, struct snf_ring_qinfo *qinfo)

  Receive and borrow many packets at once.

- int snf_ring_return_many (snf_ring_t ring, uint32_t data_qlen, struct snf_ring_qinfo *qinfo)

  Return packet space to receive ring.
- int snf_ring_getstats (snf_ring_t ringh, struct snf_ring_stats *stats)

  Get statistics from a receive ring.

5.1.1 Detailed Description

API Reference for SNF and usage model.

5.1.2 API Reference

The Sniffer API model is best summarized in the following steps:

- Initialize the API with the provided SNF_VERSION_API pre-define to ensure future ABI compatibility snf_init.
- Query available Sniffer-capable devices if required (snf_getifaddrs) and open the device by port number (snf_-open) with one or more rings.
- Have the current (or many other threads) open a receive ring for capture (snf_ring_open), then start packet capture (snf_start).
- Each ring is serviced in a loop by one or more consumers (snf_ring_recv).
- Each ring is closed and finally, the device can be closed (snf_ring_close and snf_close).
- Packets can be injected by opening injection handles, which can be opened independently from the receive capability. Injection handles can be used to re-inject packets or as a packet generator.

5.1.3 Processing Model

The API promotes a processing model mindful of the multiple cores available on today's systems by enabling packets to be received through multiple receive rings. This feature appears in many 10GBit Ethernet drivers as a Receive-Side Scaling (RSS) but instead of being an feature internal to the driver, the Sniffer API presents the rings (or queues) as a first-order interface to the user. Instead of internally maintaining multiple rings as part of the driver, the Sniffer API allows users to directly create rings and retain better control over how rings are allocated on a typical multicore system. The flexibility over ring allocation is balanced with a receive function with much stricter semantics, but only to pursue these goals:

- Allow Sniffer implementation to partition incoming network traffic through receive rings while ensuring that packets maintain order in TCP/UDP flows. This virtualizes the capture interface and allows parallel processing of incoming packet data.
- Give users "zero-copy" access to captured data packets such that users can operate directly on the packet data that was provided by the NIC.
- Minimize the use of locking and other software overheads in the critical path to achieve high packet rates (both on each ring and as an aggregate).

5.1.3.1 Multiple Receive Rings

The number of requested receive rings is part of the function call to open a device for packet capture. While users retain control over how many rings are to be allocated by the API, the API best matches processing models that allocate one ring per core, where each core will both capture packets from Sniffer and process/analyze them in place. In-place processing refers to an effort to contain the packet capture and analysis within a CPU core's memory domain, which includes all levels of caches and RAM closest to the core. Since today's multicore platforms are increasingly complex in the non-uniformity of memory access (NUMA), it is best to minimize the amount of memory accesses across memory domains and to maximize the amount of work that can be done by one core within its own memory domain. As such, Sniffer assumes that for best performance, users recognize the importance of opening rings and maintain strong ties between each ring's single consuming thread and the core where it is scheduled.

5.1.3.1.1 Receive Side Scaling Modes

When multiple rings are used, users can specify how packets should be partitioned based on a set of **RSS** flags. These flags instruct how Sniffer should hash incoming packets such that per-flow affinity is maintained in the same ring if so desired. This effectively allows each ring to virtualize the underlying network interface by ensuring that packets within the same flow are deterministically delivered to the same receive ring. Users should not need to reconstruct the ordering of flows even though incoming data is being split across many receive rings.

5.1.3.2 Zero-Copy receives

When opening a device with one or multiple rings, users can also specify combined amount of memory that can be consumed by all rings to store packet data. If left unspecified, the implementation will default to choosing a relatively conservative amount of memory, assuming that consumers can process incoming packets at line rate. Resizing the ring should be considered only to address specific buffering concerns and when multiple rings are not possible. It is possible that larger rings are necessary to mitigate the non-real-time behavior of some of the supported operating systems. Larger data rings will only serve as a temporary relief for users that cannot consume incoming data at line rates. If the application does not return packets to the data ring as fast as the packets are coming in, the receive ring will eventually overflow and the packets will be dropped.

5.1.3.3 API Software Overhead

The API tries to minimize software overhead in as many areas that can be addressed directly by the library. This includes duplicating some internal data structures to prevent false sharing between multiple ring consumers. More importantly, however, is the explicit decision to **not** provide any locking for the main receive function (snf_ring_recv). Even light forms of locking can impact processing rates when incoming data rates are roughly 15 Mpps (or every 70 nanoseconds). Instead, the API promotes the use of multiple rings to improve concurrency in packet processing and leave more breathing room for packet-per-packet analysis on each consuming core.

5.1.3.4 Thread-Safety

Users should consult the API reference for information on thread safety as each function extensively documents how in can be used in threaded environments. While most API functions provide strong thread safety guarantees, the main receive function (snf_ring_recv) is specifically not thread-safe for software overhead reasons explained in API Software Overhead. The expectation is that multiple threads would each open their own receive rings and independently consume packets on their own rings. Specifically, the Sniffer implementation assumes that with each successive call

to snf_ring_recv within the context of a ring always means that the previous packet was consumed by the previous receive on that ring. This should not come as a surprise for users that have been exposed to the serialization present in libpcap-type interfaces. However, this may not necessarily match up well with users that expected to separate their computing resources (i.e. cores) between capture and analysis. These users can always resort to using a single receive ring.

5.1.4 Implementation details

5.1.4.1 Timestamps

Packet timestamps are available from each packet for packet arrival as seen from the NIC. The 64-bit nanosecond since EPOC timestamp is returned to the user in the receive call. It should be straightforward for the user to convert nanoseconds to a **struct timeval** if so needed. There are 3 timestamping modes: Host, Timesource (Hardware) and Arista. In Host timestamping mode, the timestamp returned to the user in the receive call is normalized in host nanoseconds ctime and Sniffer internally ensures that both clocks remain synchronized at a regular interval. The frequency of the NIC's clock is 2MHz plus/minus 100ppm. Timesource (Hardware) mode is available with 10G-PCI-E2-8C2-2S-SYNC adapters when connected to an IRIG-B time source. Arista mode is available when the NIC port is connected to an Arista 71xx series switch port with "FCS append mode" timestamping enabled and the mode is also enabled in Sniffer10G with the MYRI_ARISTA_ENABLE_TIMESTAMPING=1 environment variable.

5.1.4.2 Library Memory Consumption

In order to efficiently use host memory, data rings are allocated when the device is opened as a pool of 4KB pages. At open time, users can specify the number of data rings that are allocated as well as the amount of data_ring_size that can be allocated for **all** rings. Internally, the Sniffer library requires an additional 1/32nd of the data_ring_size for each ring and some additional inter-ring synchronization memory. This can be summarized as at most 10% of the data_ring_size is allocated internally by Sniffer.

5.1.4.3 Ethernet and Sniffer Driver Modes

At any time, irrespective of whether the underlying device is enabled for Sniffer, users can configure a Sniffer-capable device as a regular ethernet device (i.e. typically via ifconfig). Unless the device is opened for capture, both send and receive functionality work as expected from any Ethernet driver. When the device is opened for capture, packets usually destined to the Ethernet driver are delivered to a Sniffer receive ring instead. However, users can still rely on the Ethernet's RAW sockets interface to send packets (a sample raw socket send enabled with Sniffer receive calls is provided as a test).

5.1.5 Define Documentation

5.1.5.1 #define SNF_VERSION_API 8

SNF API version number (16 bits) Least significant byte increases for minor backwards compatible changes in the API. Most significant byte increases for incompatible changes in the API

0x0008: Add link speed support.

0x0007: Add Multiple Application support and snf_set_app_id() function.

0x0006: Internal driver/library API changes.

0x0005: Internal driver/library API changes.

0x0004: Add more injection support and aggregate port opens

0x0003: Add injection support and 3 send counters in statistics.

0x0002: Add nic_bytes_recv counter to stats to help users calculate the amount of bandwidth that is actually going through the NIC port.

5.1.6 Typedef Documentation

5.1.6.1 typedef struct snf_handle* snf_handle_t

opaque snf device handle

Opaque snf handle, allocated at snf_open time when a device can be successfully opened

5.1.6.2 typedef struct snf_ring* snf_ring_t

opaque snf ring handle

Opaque snf ring handle, allocated at snf_ring_open time when a ring can be succesfully opened. The ring itself can represent a single or an aggregate of physical rings.

5.1.7 Enumeration Type Documentation

5.1.7.1 enum snf_link_state

Link state enumeration, returned by snf_get_link_state

5.1.7.2 enum snf_timesource_state

Timesource state (for -SYNC NICs), returned by snf_get_timesource_state

Enumerator:

SNF_TIMESOURCE_LOCAL Local timesource (no external). Returned if there is no available external timesource or if its use was explicitly disabled.

SNF_TIMESOURCE_EXT_UNSYNCED External Timesource: not synchronized (yet)

SNF_TIMESOURCE_EXT_SYNCED External Timesource: synchronized

SNF_TIMESOURCE_EXT_FAILED External Timesource: NIC failure to connect to source

SNF_TIMESOURCE_ARISTA_ACTIVE Arista switch is sending ptp timestamps

SNF_TIMESOURCE_PPS PPS is being used for time

5.1.8 Function Documentation

5.1.8.1 int snf_close (snf_handle_t *devhandle*)

Close port.

This function can be closed once all opened rings (if any) are closed through snf_ring_close. Once a port is determined to be closable, it is implicitly called as if a call had been previously made to snf_stop.

Return values

0	Successful.
EBUSY	Some rings are still opened and the port cannot be closed (yet).

Postcondition

If successful, all resources allocated at open time are unallocated and the device switches from Sniffer mode to Ethernet mode such that the Ethernet driver resumes receiving packets.

5.1.8.2 void snf_freeifaddrs (struct snf_ifaddrs * *ifaddrs*)

Free the list of library allocated memory for snf_getifaddrs

Parameters

ifaddrs	Pointer to ifaddrs allocated via snf_getifaddrs

5.1.8.3 int snf_get_link_speed (snf_handle_t *devhandle*, uint64_t * *speed*)

Get link speed on opened handle

Parameters

devhandle	Device handle
speed	Returns speed in bits-per-second for the link

Remarks

The cost of retrieving the link speed requires a function call that reads information kept in kernel host memory (i.e. no PCI bus reads).

5.1.8.4 int snf_get_link_state (snf_handle_t *devhandle*, enum snf_link_state * *state*)

Get link status on opened handle

Parameters

devhandle	Device handle
state	Returns one of SNF_LINK_DOWN or SNF_LINK_UP

Remarks

The cost of retrieving the link state requires a function call that reads state kept in kernel host memory (i.e. no PCI bus reads).

5.1.8.5 int snf_get_timesource_state (snf_handle_t *devhandle*, enum snf_timesource_state * *state*)

Get Timesource information from opened handle

Parameters

devhandle	Device handle
state	Returns one of snf_timesource_state

Remarks

The cost of retrieving the timesource state requires a function call that reads state kept in kernel host memory (i.e. no PCI bus reads).

5.1.8.6 int snf_getifaddrs (struct snf_ifaddrs ** *ifaddrs_o*)

Get a list of Sniffer-capable ethernet devices.

Parameters

ifaddrs_o	Library-allocated list of Sniffer-capable devices

Remarks

Much like getifaddrs, the user can traverse the list until snf_ifa_next is NULL. The interface will show up if the ethernet driver sees the device but the interface does not have to be brought up with an IP address (i.e. no need to 'ifconfig up').

Postcondition

User should call snf_freeifaddrs to free the memory that was allocated by the library.

5.1.8.7 int snf_getportmask_linkup (uint32_t * *mask_o*, int * *cnt_o*)

Get a mask of all Sniffer-capable ports that have their link state set to UP

The least significant bit represents port 0.

Similar to snf_getportmask_valid except that only ports with an active link are set in the mask.

Parameters

mask_o	bitmask set at output
cnt_o	Number of bits set in bitmask

Return values

0	Successful.
ENODEV	Error obtaining port information

5.1.8.8 int snf_getportmask_valid (uint32_t * *mask_o*, int * *cnt_o*)

Get a mask of all Sniffer-capable ports.

The least significant bit represents port 0.

Parameters

mask_o	bitmask set at output
cnt_o	Number of bits set in bitmask

Return values

0	Successful.
ENODEV	Error obtaining port information

5.1.8.9 int snf_init (uint16_t *api_version*)

Initialize Sniffer Library with api_version == SNF_VERSION_API.

Initializes the sniffer library.

Parameters

api_version	Must always be SNF_VERSION_API

Remarks

This must be called before any other call to the sniffer library.
The library may be safely initialized multiple times, although the api_version should be the same SNF_VERSION_API each time.

5.1.8.10 int snf_open (uint32_t *portnum*, int *num_rings*, const struct snf_rss_params * *rss_params*, int64_t *dataring_sz*, int *flags*, snf_handle_t * *devhandle*)

Open device for single or multi-ring operation.

Opens a port for sniffing and allocates a device handle.

Parameters

portnum	Port numbers can be interpreted as integers for a specific port number or as a mask when SNF_F_AGGREGATE_PORTMASK is specified in flags. Port information can be obtained through snf_getifaddrs and active/valid masks are available with snf_getportmask_valid and snf_getportmask_linkup. As a special case, if portnum -1 is passed, the library will internally open a portmask as if snf_getportmask_valid was called.
num_rings	Number of rings to allocate for receive-side scaling feature, which determines how many different threads can open their own ring via snf_ring_open(). If set to 0 or less than zero, default value is used unless SNF_NUM_RINGS is set in the environment.

rss_params	Points to a user-initialized structure that selects the RSS mechanism to apply to each incoming packet. This parameter is only meaningful if there are more than 1 rings to be opened. By default, if users pass a NULL value, the implementation will select its own mechanism to divide incoming packets across rings. RSS parameters are documented in Receive-Side Scaling (RSS).
dataring_sz	Represents the total amount of memory to be used to store incoming packet data for *all* rings to be opened. If the value is set to 0 or less than 0, the library tries to choose a sensible default unless SNF_DATARING_SIZE is set in the environment. The value can be specified in megabytes (if it is less than 1048576) or is otherwise considered to be in bytes. In either case, the library may slightly adjust the user's request to satisfy alignment requirements (typically 2MB boundaries).
flags	A mask of flags documented in Open flags for process-sharing, port aggregation and packet duplication.
devhandle	Device handle allocated if the call is successful

Return values

0	Successful. the port is opened and a value devhandle is allocated (see remarks)
EBUSY	Device is already opened
EINVAL	Invalid argument passed, most probably num_rings (if not, check syslog)
E2BIG	Driver could not allocate requested dataring_sz (check syslog)
ENOMEM	Either library or driver did not have enough memory to allocate handle descriptors (but not data ring).
ENODEV	Device portnum can't be opened

Postcondition

If successful, the NIC switches from Ethernet mode to Capture mode and the Ethernet driver stops receiving packets.
If successful, a call to snf_start is required to the Sniffer-mode NIC to deliver packets to the host, and this call must occur after at least one ring is opened (snf_ring_open).

5.1.8.11 int snf_open_defaults (uint32_t *portnum*, snf_handle_t * *devhandle*)

Open device for single or multi-ring operation.

Opens a port for sniffing and allocates a device handle using system defaults.

This function is a simplified version of snf_open and ensures that the resulting device is opened according to system defaults. Since the number of rings and flags can be set by module parameters, some installations may prefer to control device-level parameters in a system-wide configuration and keep the library calls simple.

This call is equivalent to

```
snf_open(portnum, 0, NULL, 0, -1, devhandle);
```

Parameters

portnum	Ports are numbered from 0 to N-1 where 'N' is the number of Myricom ports available on the system. snf_getifaddrs() may be a useful utility to retrieve the port number by interface name or mac address if there are multiple
devhandle	Device handle allocated if the call is successful

See also

snf_open

5.1.8.12 int snf_ring_close (snf_ring_t *ringh*)

Close a ring

This function is used to inform the underlying device that no further calls to snf_ring_recv will be made. If the device is not subsequently closed (snf_close), all packets that would have been delivered to this ring are dropped. Also, by calling this function, users confirm that all packet processing for packets obtained on this ring via snf_ring_recv is complete.

Parameters

ringh	Ring handle

Return values

0	Successful.

Postcondition

The user has processed the last packet obtained with snf_ring_recv and the device can safely be closed via snf_close if all other rings are also closed.

5.1.8.13 int snf_ring_getstats (snf_ring_t *ringh*, struct snf_ring_stats * *stats*)

Get statistics from a receive ring.

Parameters

ringh	Ring handle
stats	User-provided pointer to a statistics structure snf_ring_stats, filled in by the library.

Remarks

This call is provided as a convenience and should not be relied on for time-critical applications or for high levels of accuracy. Statistics are only updated by the NIC periodically.

Warning

Administrative clearing of NIC counters while a Sniffer-based application is running may cause some of the counters to be incorrect.

5.1.8.14 int snf_ring_open (snf_handle_t *devhandle*, snf_ring_t * *ringh*)

Opens the next available ring

Parameters

devhandle	Device handle, obtained from a successful call to snf_open
ringh	Ring handle allocated if the call is successful.

Return values

0	Successful. The ring is opened and ringh contains the ring handle.
EBUSY	Too many rings already opened

Remarks

This function will consider the value of the SNF_RING_ID environment variable. For more control over ring allocation, consider using snf_ring_open_id instead.

Postcondition

If successful, a call to snf_start is required to the Sniffer-mode NIC to deliver packets to the host.

5.1.8.15 int snf_ring_open_id (snf_handle_t *devhandle*, int *ring_id*, snf_ring_t * *ringh*)

Opens a ring from an opened port.

Parameters

devhandle	Device handle, obtained from a successful call to snf_open
ring_id	Ring number to open, from 0 to **num_rings** - 1. If the value is -1, this function behaves as if snf_ring_open was called.
ringh	Ring handle allocated if the call is successful.

Return values

0	Successful. The ring is opened and ringh contains the ring handle.
EBUSY	If ring_id == -1, Too many rings already opened. If ring_id >= 0, that ring is already opened.

Remarks

Unlike snf_ring_open this function ignores the environment variable SNF_RING_ID since the expectation is that users want to directly control ring allocation (unlike through libpcap).

Postcondition

If successful, a call to snf_start is required to the Sniffer-mode NIC to deliver packets to the host.

5.1.8.16 int snf_ring_portinfo (snf_ring_t *ring*, struct snf_ring_portinfo * *portinfo*)

Returns information for the ring. For aggregated rings, returns information for each of the physical rings. It is up to the user to make sure they have allocated enough memory to hold the information for all the physical rings in an aggregated ring.

Parameters

ring	Ring handle (from snf_ring_open)
portinfo	Pointer to memory allocated by the user that will be filled in with the information.

5.1.8.17 int snf_ring_recv (snf_ring_t *ringh*, int *timeout_ms*, struct snf_recv_req * *recv_req*)

Receive next packet from a receive ring.

This function is used to return the next available packet in a receive ring. The function can block indefinitely, for a specific timeout or be used as a non-blocking call with a timeout of 0.

Parameters

ringh	Ring handle (from snf_ring_open)
timeout_ms	Receive timeout to control how the function blocks for the next packet. If the value is less than 0, the function can block indefinitely. If the value is 0, the function is guaranteed to never enter a blocking state and returns EAGAIN unless there is a packet waiting. If the value is greater than 0, the caller indicates a desired wait time in milliseconds. With a non-zero wait time, the function only blocks if there are no outstanding packets. If the timeout expires before a packet can be received, the function returns EAGAIN (and **not** ETIMEDOUT). In all cases, users should expect that the function may return EINTR as the result of signal delivery.
recv_req	Receive Packet structure, only updated when a the function returns 0 for a successful packet receive (snf_recv_req)

Return values

0	Successful packet delivery, recv_req is updated with packet information.
EINTR	The call was interrupted by a signal handler
EAGAIN	No packets available (only when timeout is >= 0).

Remarks

The packet returned always points directly into the receive ring where the NIC has DMAed the packet (there are no copies). As such, the user obtains a pointer to library/driver allocated memory. Users can modify the contents of the packets but should remain within the boundaries of **pkt_addr** and **length**.
Upon calling the function, the library assumes that the user is done processing the previous packet. The same assumption is made when the ring is closed (snf_ring_close).

5.1.8.18 int snf_ring_recv_many (snf_ring_t *ring*, int *timeout_ms*, struct snf_recv_req * *req_vector*, int *nreq_in*, int * *nreq_out*, struct snf_ring_qinfo * *qinfo*)

Receive and borrow many packets at once.

This function allows callers to receive one or more packets per call. Contrary to snf_ring_recv, this function assumes that callers will split the functionality to receive packets (or borrow them) and the functionality to return packets through snf_ring_return_many.

Parameters

ring	Ring handle (from snf_ring_open)

timeout_ms	Receive timeout to control how the function blocks for the next packet. See complete documentation in snf_ring_recv.
req_vector	Vector of receive packet structures provided by the user and only updated when a packet is received.
nreq_in	Number of receive packet structures provided in **req_vector**. No more than **nreq_in** packets can be received.
nreq_out	Output value for the number of packets actually received and updated in **req_vector**
qinfo	If non-NULL, the qinfo structure is updated before the function returns 0 or EAGAIN (the function is not updated for other error conditions). `// See snf_ring_return_many documentation for examples`

5.1.8.19 int snf_ring_return_many (snf_ring_t *ring*, uint32_t *data_qlen*, struct snf_ring_qinfo * *qinfo*)

Return packet space to receive ring.

Under the borrow-many-return-many receive model, it is up to the user to return space in the receive ring. The user achieves this by accumulating packet lengths from the **length_data** parameter from each packet received and returning the space through this function call.

Parameters

ring	Ring handle (from snf_ring_open)
data_qlen	Amount of data returned by previously consumed packets. As a special case, if the value -1 is provided, all data previously borrowed through snf_ring_recv_many will be returned.
qinfo	If non-NULL, the qinfo structure is updated before the function returns.

```
// Example that shows how the borrow-many-return-many receive model
// works.  Since the underlying data is a ring, we don't return a
// packet count, we return data space.  The library function call
// overhead can be amortized for high packet rates.
void
recv_dispatch(snf_ring_t ringh, void (*handler)(void *pkt, uint32_t length)
{
   struct snf_recv_req reqs[32];
   int rc, nreqs;
   int i, wait_msec = 1000;
   extern int do_exit;
   uint32_t return_length = 0;

   while (1) {
      // Wait up to 1 second for at least one packet to arrive with a
      // maximum of 32 packets within a single call
      rc = snf_ring_recv_many(wrk->hring, wait_msec, reqs, 32, &nreqs, NULL);
      if (rc == 0) {
        for (i = 0; i < nreqs; i++) {
          // We handle each packet separately and accumulate the amount
          // of data each packet consumes in the ring. Because of
          // alignments length_data is somewhat larger than the packet
          // length
          handler(reqs[i].pkt_addr, reqs[i].length);
          return_length += reqs[i].length_data;
        }
        // We return the data in a single call.  We could have gathered
```

```
        // some queue information but not this time around (qinfo set
        // to NULL)
        rc = snf_ring_return_many(wrk->hring, return_length, NULL);
        assert(rc == 0);
        return_length = 0;
      }
      else if (rc == EINTR)
        if (do_exit)
          break;
    }
  }
```

5.1.8.20 int snf_set_app_id (int32_t *id*)

Set the application ID.

Sets the application ID.

The user may set the application ID after the call to snf_init, but before snf_open. When the application ID is set, Sniffer duplicates receive packets to multiple applications. Each application must have a unique ID. Then, each application may utilize a different number of rings. The application can be a process with multiple rings and threads. In this case all rings have the same ID. Or, multiple processes may share the same application ID.

The user may store the application ID in the environment variable SNF_APP_ID, instead of calling this function. Both actions have the same effect. SNF_APP_ID overrides the ID set via snf_set_app_id.

The user may not run a mix of processes with valid application IDs (not -1) and processes with no IDs (-1). Either all processes have valid IDs or none of them do.

Parameters

id	A 32-bit signed integer representing the application ID. A valid ID is any value except -1. -1 is reserved and represents "no ID".

Return values

0	Successful
EINVAL	snf_init has not been called or id is -1.

5.1.8.21 int snf_start (snf_handle_t *devhandle*)

Start packet capture on a port. Packet capture is only started if it is currently stopped or has not yet started for the first time.

Parameters

devhandle	Device handle

Remarks

It is safe to restart packet capture via snf_start and snf_stop.
This call must be called before any packet can be received.

5.1.8.22 int snf_stop (snf_handle_t *devhandle*)

Stop packet capture on a port. This function should be used carefully in multi-process mode as a single stop command stops packet capture on all rings. It is usually best to simply snf_ring_close a ring to stop capture on a ring.

Parameters

devhandle	Device handle

Remarks

Stop instructs the NIC to drop all packets until the next snf_start() or until the port is closed. The NIC only resumes delivering packets when the port is closed, not when traffic is stopped.

5.2 Receive-Side Scaling (RSS)

Data Structures

- struct snf_rss_mode_function
- struct snf_rss_params

Defines

- #define SNF_RSS_IPV4 SNF_RSS_IP

Enumerations

- enum snf_rss_params_mode { SNF_RSS_FLAGS = 0, SNF_RSS_FUNCTION = 1 }
- enum snf_rss_mode_flags {
  SNF_RSS_IP = 0x01, SNF_RSS_SRC_PORT = 0x10, SNF_RSS_DST_PORT = 0x20, SNF_RSS_GTP = 0x40,
  SNF_RSS_GRE = 0x80 }

5.2.1 Detailed Description

These options can be passed as parameters to snf_open when RSS is used.

5.2.2 Define Documentation

5.2.2.1 #define SNF_RSS_IPV4 SNF_RSS_IP

Alias for SNF_RSS_IP since IPv4 and IPv6 are always both enabled

5.2.3 Enumeration Type Documentation

5.2.3.1 enum snf_rss_mode_flags

RSS parameters for SNF_RSS_FLAGS, flags that can be specified to let the implementation know which fields are significant when generating the hash. By default, RSS is computed on IPv4/IPv6 addresses and source/destination ports when the protocol is TCP or UDP or SCTP, the equivalent of which would be

```
struct snf_rss_params rssp;
rssp.mode = SNF_RSS_FLAGS;
rssp.params.rss_flags = SNF_RSS_IP | SNF_RSS_SRC_PORT | SNF_RSS_DST_PORT;

snf_handle_t hsnf;
int rc = snf_open(0, 0, &rssp, 0, -1, &hsnf);
if (rc == 0)
  printf("RSS will by applied to IP addresses and TCP/UDP ports if applicable"
     );
```

Enumerator:

SNF_RSS_IP Include IP (v4 or v6) SRC/DST addr in hash

SNF_RSS_SRC_PORT Include TCP/UDP/SCTP SRC port in hash

SNF_RSS_DST_PORT Include TCP/UDP/SCTP DST port in hash

SNF_RSS_GTP Include GTP TEID in hash

SNF_RSS_GRE Include GRE contents in hash

5.2.3.2 enum snf_rss_params_mode

RSS select mode

Enumerator:

SNF_RSS_FLAGS Apply RSS using specified flags

SNF_RSS_FUNCTION Apply RSS using user-defined function: Kernel API only

5.3 Open flags for process-sharing, port aggregation and packet duplication

Defines

- #define SNF_F_PSHARED 0x1
- #define SNF_F_AGGREGATE_PORTMASK 0x2
- #define SNF_F_RX_DUPLICATE 0x300

5.3.1 Detailed Description

These options are passed to the **flags** parameter in snf_open to enable various receive mechanisms.

5.3.2 Define Documentation

5.3.2.1 #define SNF_F_AGGREGATE_PORTMASK 0x2

Device can be opened for port aggregation (or merging). When this flag is passed, the **portnum** parameter in snf_open is interpreted as a bitmask where each set bit position represents a port number. The Sniffer library will then attempt to open every portnum with its bit set in order to merge the incoming data to the user from multiple ports. Subsequent calls to snf_ring_open return a ring handle that internally opens a ring on all underlying ports.

5.3.2.2 #define SNF_F_PSHARED 0x1

Device can be process-sharable. This allows multiple independent processes to share rings on the capturing device. This option can be used to design a custom capture solution but is also used in libpcap when multiple rings are requested. In this scenario, each libpcap device sees a fraction of the traffic if multiple rings are used unless the SNF_F_RX_DUPLICATE option is used, in which case each libpcap device sees the same incoming packets.

5.3.2.3 #define SNF_F_RX_DUPLICATE 0x300

Device can duplicate packets to multiple rings as opposed to applying RSS in order to split incoming packets across rings. Users should be aware that with N rings opened, N times the link bandwidth is necessary to process incoming packets without drops. The duplication happens in the host rather than the NIC, so while only up to 10Gbits of traffic crosses the PCIe, N times that bandwidth is necessary on the host.

When duplication is enabled, RSS options are ignored since every packet is delivered to every ring.

5.4 Packet injection

Data Structures

- struct snf_pkt_fragment

  Fragment for snf_inject_send_v.
- struct snf_inject_stats

Typedefs

- typedef struct snf_inject_handle * snf_inject_t

Functions

- int snf_inject_open (int portnum, int flags, snf_inject_t *handle)

  Open a port for injection and allocate an injection handle.
- int snf_get_injection_speed (snf_inject_t devhandle, uint64_t *speed)
- int snf_inject_send (snf_inject_t inj, int timeout_ms, int flags, const void *pkt, uint32_t length)

  Send a packet and optionally block until send resources are available.
- int snf_inject_sched (snf_inject_t inj, int timeout_ms, int flags, const void *pkt, uint32_t length, uint64_t delay_ns)

  Send a packet with hardware delay and optionally block until send resources are available.
- int snf_inject_send_v (snf_inject_t inj, int timeout_ms, int flags, struct snf_pkt_fragment *frags_vec, int nfrags, uint32_t length_hint)

  Send a packet assembled from a vector of fragments and optionally block until send resources are available.
- int snf_inject_sched_v (snf_inject_t inj, int timeout_ms, int flags, struct snf_pkt_fragment *frags_vec, int nfrags, uint32_t length_hint, uint64_t delay_ns)

  Send a packet assembled from a vector of fragments at a scheduled point relative to the start of the prior packet and optionally block until send resources are available.
- int snf_inject_close (snf_inject_t inj)

  Close injection handle.
- int snf_inject_getstats (snf_inject_t inj, struct snf_inject_stats *stats)

  Get statistics from an injection handle.

5.4.1 Detailed Description

SNF Packet injection routines that can be used for independent packet generation or coupled to reinject packets received with snf_ring_recv.

5.4.2 Typedef Documentation

5.4.2.1 typedef struct snf_inject_handle* snf_inject_t

Opaque injection handle, allocated by snf_inject_open. There are only a limited amount of injection handles per NIC/port.

5.4.3 Function Documentation

5.4.3.1 int snf_get_injection_speed (snf_inject_t *devhandle*, uint64_t * *speed*)

Get link speed on opened injection handle

Parameters

devhandle	Device handle
speed	Returns speed in bits-per-second for the link

Remarks

The cost of retrieving the link speed requires a function call that reads information kept in kernel host memory (i.e. no PCI bus reads).

5.4.3.2 int snf_inject_close (snf_inject_t *inj*)

Close injection handle.

This function closes an injection handle and ensures that all pending sends are sent by the NIC.

Parameters

inj	Injection handle

Return values

0	Successful

Postcondition

Once closed, the injection handle will have ensured that any pending sends have been sent out on the wire. The handle is then made available again for the underlying port's limited amount of handles.

5.4.3.3 int snf_inject_getstats (snf_inject_t *inj*, struct snf_inject_stats * *stats*)

Get statistics from an injection handle.

Parameters

inj	Injection Handle
stats	User-provided pointer to a statistics structure snf_inject_stats, filled in by the SNF implementation.

Remarks

This call is provided as a convenience and should not be relied on for time-critical applications or for high levels of accuracy. Statistics are only updated by the NIC periodically.

Warning

Administrative clearing of NIC counters while a Sniffer-based application is running may cause some of the counters to be incorrect.

5.4.3.4 int snf_inject_open (int *portnum*, int *flags*, snf_inject_t * *handle*)

Open a port for injection and allocate an injection handle.

Parameters

portnum	Ports are numbered from 0 to N-1 where 'N' is the number of Myricom ports available on the system. snf_getifaddrs() may be a useful utility to retrieve the port number by interface name or mac address if there are multiple
flags	Flags for injection handle. None are currently defined.
handle	Injection handle allocated if the call is successful.

Return values

0	Success. An injection handle is opened and allocated.
EBUSY	Ran out of injection handles for this port
ENOMEM	Ran out of memory to allocate new injection handle

5.4.3.5 int snf_inject_sched (snf_inject_t *inj*, int *timeout_ms*, int *flags*, const void * *pkt*, uint32_t *length*, uint64_t *delay_ns*)

Send a packet with hardware delay and optionally block until send resources are available.

This send function is used for paced packet injection. This function can be used as part of a packet replay program. When the function returns successfully, the packet is guaranteed to be completely buffered by SNF: no references are kept to the input data and the caller is free to safely modify its contents. The SNF implementation delays transmitting the packet according to the delay_ns parameter, relative to the start of the prior packet.

Parameters

inj	Injection handle
timeout_ms	Timeout in milliseconds to wait if insufficient send resources are available to inject a new packet. Insufficient resources can be a lack of send descriptors or a full send queue ring. If timeout_ms is 0, the function won't block for send resources and returns EAGAIN.
flags	Flags (currently none).
pkt	Pointer to the packet to be sent. The packet must be a pointer to a complete Ethernet frame (without the trailing CRC) and start with a valid Ethernet header. The hardware will append 4-CRC bytes at the end of the packet. The maximum valid packet size is 9000 bytes and is enforced by the library. The minimum valid packet size is 60 bytes, although any packet smaller than 60 bytes will be accepted by the library and padded by the hardware.
length	The length of the packet, excluding the trailing 4 CRC bytes.
delay_ns	The minimum delay between the start of the prior packet and the start of this packet. Packets with a delay less than the time to send the prior packet are send immediately. It is recommened to use 0 as the delta on the first packet sent.

Return values

0	Successful. The packet is buffered by SNF.
EAGAIN	Insufficient resources to send packet. If timeout_ms is non-zero, the caller will have blocked at least that many milliseconds before resources could become available.
EINVAL	Packet length is larger than 9000 bytes.
ENOTSUP	The hardware does not support injection pacing.

Postcondition

If successful, the packet is completely buffered for sending by SNF. The implementation guarantees that it will eventually send the packet out, as scheduled, without requiring further calls into SNF.

5.4.3.6 int snf_inject_sched_v (snf_inject_t *inj*, int *timeout_ms*, int *flags*, struct snf_pkt_fragment * *frags_vec*, int *nfrags*, uint32_t *length_hint*, uint64_t *delay_ns*)

Send a packet assembled from a vector of fragments at a scheduled point relative to the start of the prior packet and optionally block until send resources are available.

This send function follows the same semantics as snf_inject_send except that the packet to be injected can be assembled from multiple fragments (or buffers).

Parameters

inj	Injection handle
timeout_ms	Timeout in milliseconds to wait if insufficient send resources are available to inject a new packet. Insufficient resources can be a lack of send descriptors or a full send queue ring. If timeout_ms is 0, the function won't block for send resources and returns EAGAIN.
flags	Flags (currently none).
frags_vec	Pointer to a vector of 1 or more buffers/fragments that can be used to compose a complete Ethernet frame (not including the trailing CRC header). The first fragment must point to a valid Ethernet header and the hardware will append its own (valid 4-byte CRC) at the end of the last buffer/fragment passed in the frags_vec. When all the fragments are added up, the maximum valid packet size is 9000 bytes and is enforced by the library. The minimum valid packet size is 60 bytes, although any packet smaller than 60 bytes will be accepted by the library and padded by the hardware.
nfrags	Number of elements in the io vector
length_hint	If non-zero, the amount is expected to be the sum of all the lengths passed in the io vector. This parameters can help the library account for space when injecting packets.
delay_ns	The minimum delay between the start of the prior packet and the start of this packet. Packets with a delay less than the time to send the prior packet are send immediately. It is recommened to use 0 as the delta on the first packet sent.

Return values

0	Successful. The packet is buffered by SNF.
EAGAIN	Insufficient resources to send packet. If timeout_ms is non-zero, the caller will have blocked at least that many milliseconds before resources could become available.
EINVAL	Packet length is larger than 9000 bytes.
ENOTSUP	The hardware does not support injection pacing.

Postcondition

If successful, the packet is completely buffered for send by SNF. The implementation guarantees that it will eventually send the packet out in a timely fashion without requiring further calls into SNF.

```
// Example that takes an existing packet and prepends the existing
// ethernet type with a vlan header.
//
int
send_prepend__vlan_tag(uint16_t vtag, void *pkt, uint32_t len)
{
  uint32_t vlanhdr = htonl(0x8100 << 16 | vtag);
  struct snf_pkt_fragment vec[3];

  // We assume that the input 'pkt' does not already contain a vlan tag
  // and that the pkt is not terminated with a CRC.  The hardware will
  // add the CRC.  We also use no timeout in the send meaning that the
  // send may return EAGAIN if there are insufficient resources to
  // queue the send.

  vec[0].ptr = (void *) pkt;
  vec[0].length = 12; // dest and src mac
  vec[1].ptr = &vlanhdr;
  vec[1].length = sizeof(vlanhdr);
  vec[2].ptr = (void *)((uint8_t *)pkt + 12);
  vec[2].length = len - 12;
  len += sizeof(vlanhdr);

  // Schedule the packet to be sent 3 us. from the last.
  return snf_inject_sched_v(hinj, 0, 0, vec, 3, len, 3000);
}
```

5.4.3.7 int snf_inject_send (snf_inject_t *inj*, int *timeout_ms*, int *flags*, const void * *pkt*, uint32_t *length*)

Send a packet and optionally block until send resources are available.

This send function is optimized for high packet rate injection. While it can be coupled with a receive ring to reinject a packet, it is not strictly necessary. This function can be used as part of a packet generator. When the function returns successfully, the packet is guaranteed to be completely buffered by SNF: no references are kept to the input data and the caller is free to safely modify its contents. A successful return does not, however, guarantee that the packet has been injected into the network. The SNF implementation may choose to hold on to the packet for coalescing in order to improve packet throughput.

Parameters

inj	Injection handle
timeout_ms	Timeout in milliseconds to wait if insufficient send resources are available to inject a new packet. Insufficient resources can be a lack of send descriptors or a full send queue ring. If timeout_ms is 0, the function won't block for send resources and returns EAGAIN.
flags	Flags (currently none).
pkt	Pointer to the packet to be sent. The packet must be a pointer to a complete Ethernet frame (without the trailing CRC) and start with a valid Ethernet header. The hardware will append 4-CRC bytes at the end of the packet. The maximum valid packet size is 9000 bytes and is enforced by the library. The minimum valid packet size is 60 bytes, although any packet smaller than 60 bytes will be accepted by the library and padded by the hardware.
length	The length of the packet, excluding the trailing 4 CRC bytes.

Return values

0	Successful. The packet is buffered by SNF.
EAGAIN	Insufficient resources to send packet. If timeout_ms is non-zero, the caller will have blocked at least that many milliseconds before resources could become available.
EINVAL	Packet length is larger than 9000 bytes.

Postcondition

If successful, the packet is completely buffered for sending by SNF. The implementation guarantees that it will eventually send the packet out in a timely fashion without requiring further calls into SNF.

5.4.3.8 int snf_inject_send_v (snf_inject_t *inj*, int *timeout_ms*, int *flags*, struct snf_pkt_fragment * *frags_vec*, int *nfrags*, uint32_t *length_hint*)

Send a packet assembled from a vector of fragments and optionally block until send resources are available.

This send function follows the same semantics as snf_inject_send except that the packet to be injected can be assembled from multiple fragments (or buffers).

Parameters

inj	Injection handle
timeout_ms	Timeout in milliseconds to wait if insufficient send resources are available to inject a new packet. Insufficient resources can be a lack of send descriptors or a full send queue ring. If timeout_ms is 0, the function won't block for send resources and returns EAGAIN.
flags	Flags (currently none).
frags_vec	Pointer to a vector of 1 or more buffers/fragments that can be used to compose a complete Ethernet frame (not including the trailing CRC header). The first fragment must point to a valid Ethernet header and the hardware will append its own (valid 4-byte CRC) at the end of the last buffer/fragment passed in the frags_vec. When all the fragments are added up, the maximum valid packet size is 9000 bytes and is enforced by the library. The minimum valid packet size is 60 bytes, although any packet smaller than 60 bytes will be accepted by the library and padded by the hardware.
nfrags	Number of elements in the io vector
length_hint	If non-zero, the amount is expected to be the sum of all the lengths passed in the io vector. This parameters can help the library account for space when injecting packets.

Return values

0	Successful. The packet is buffered by SNF.
EAGAIN	Insufficient resources to send packet. If timeout_ms is non-zero, the caller will have blocked at least that many milliseconds before resources could become available.
EINVAL	Packet length (in length_hint or the sum of all frags_vec lens) is larger than 9000 bytes.

Postcondition

If successful, the packet is completely buffered for send by SNF. The implementation guarantees that it will eventually send the packet out in a timely fashion without requiring further calls into SNF.

```
// Example that takes an existing packet and prepends the existing
```

```
// ethernet type with a vlan header.
//
int
send_prepend__vlan_tag(uint16_t vtag, void *pkt, uint32_t len)
{
  uint32_t vlanhdr = htonl(0x8100 << 16 | vtag);
  struct snf_pkt_fragment vec[3];

  // We assume that the input 'pkt' does not already contain a vlan tag
  // and that the pkt is not terminated with a CRC.  The hardware will
  // add the CRC.  We also use no timeout in the send meaning that the
  // send may return EAGAIN if there are insufficient resources to
  // queue the send.

  vec[0].ptr = (void *) pkt;
  vec[0].length = 12; // dest and src mac
  vec[1].ptr = &vlanhdr;
  vec[1].length = sizeof(vlanhdr);
  vec[2].ptr = (void *)((uint8_t *)pkt + 12);
  vec[2].length = len - 12;
  len += sizeof(vlanhdr);

  return snf_inject_send_v(hinj, 0, 0, vec, 3, len);
}
```

5.5 Packet reflect to netdev (kernel stack)

Typedefs

- typedef void * snf_netdev_reflect_t

Functions

- int snf_netdev_reflect_enable (snf_handle_t hsnf, snf_netdev_reflect_t *handle)

  Enable a network device for packet reflection.
- int snf_netdev_reflect (snf_netdev_reflect_t ref_dev, const void *pkt, uint32_t length)

  Reflect a packet to the network device.

5.5.1 Detailed Description

Network packets acquired through Sniffer can be reflected back into the kernel path as if the device had initially sent then through to the regular network stack.

While Sniffer users are typically expected to process a significant portion of their packets with less overhead in userspace, this feature is provided as a convenience to allow some packets to be processed back in the kernel. The implementation makes no explicit step to make the kernel-based processing any faster than it is when Sniffer is not being used (in fact, it is probably much slower).

5.5.2 Typedef Documentation

5.5.2.1 typedef void* snf_netdev_reflect_t

Opaque handle returned by snf_netdev_reflect_enable and used to reflect packets onto by snf_netdev_reflect.

5.5.3 Function Documentation

5.5.3.1 int snf_netdev_reflect (snf_netdev_reflect_t *ref_dev*, const void * *pkt*, uint32_t *length*)

Reflect a packet to the network device.

Parameters

ref_dev	Reflection handle
pkt	Pointer to the packet to be reflected to the network device. The packet must be a pointer to a complete Ethernet frame (without the trailing CRC) and start with a valid Ethernet header.
length	The length of the packet, excluding the trailing 4 CRC bytes.

Return values

0	Successful. The packet is buffered by SNF.

Postcondition

If succcessful, the packet is completely buffered into the network device receive path.

5.5.3.2 int snf_netdev_reflect_enable (snf_handle_t *hsnf,* snf_netdev_reflect_t * *handle*)

Enable a network device for packet reflection.

Parameters

hsnf	handle for network device to reflect onto, obtained by snf_open
handle	Reflection handle.

Return values

0	Success. An reflection handle is enabled.

Chapter 6

Namespace Documentation

6.1 snf Namespace Reference

6.1.1 Detailed Description

Sniffer

Author

Myricom, Inc.

Chapter 7

Data Structure Documentation

7.1 snf_ifaddrs Struct Reference

Data Fields

- struct snf_ifaddrs * snf_ifa_next
- const char * snf_ifa_name
- uint32_t snf_ifa_portnum
- int snf_ifa_maxrings
- uint8_t snf_ifa_macaddr [6]
- uint8_t pad [2]
- int snf_ifa_maxinject
- enum snf_link_state snf_ifa_link_state
- uint64_t snf_ifa_link_speed

7.1.1 Detailed Description

Structure to map Interfaces to Sniffer port numbers

7.1.2 Field Documentation

7.1.2.1 uint8_t snf_ifaddrs::pad[2]

Internal padding (ignore)

7.1.2.2 uint64_t snf_ifaddrs::snf_ifa_link_speed

Link Speed (bps)

7.1.2.3 enum snf_link_state snf_ifaddrs::snf_ifa_link_state

Underlying port's state (DOWN or UP)

7.1.2.4 uint8_t snf_ifaddrs::snf_ifa_macaddr[6]

MAC address

7.1.2.5 int snf_ifaddrs::snf_ifa_maxinject

Maximum TX injection handles supported

7.1.2.6 int snf_ifaddrs::snf_ifa_maxrings

Maximum RX rings supported

7.1.2.7 const char* snf_ifaddrs::snf_ifa_name

interface name, as in ifconfig

7.1.2.8 struct snf_ifaddrs* snf_ifaddrs::snf_ifa_next

next item or NULL if last

7.1.2.9 uint32_t snf_ifaddrs::snf_ifa_portnum

snf port number

7.2 snf_inject_stats Struct Reference

Data Fields

- uint64_t inj_pkt_send
- uint64_t nic_pkt_send
- uint64_t nic_bytes_send

7.2.1 Detailed Description

Structure to return statistics from an injection handle. The hardware-specific counters (nic_) apply to all injection handles.

7.2.2 Field Documentation

7.2.2.1 uint64_t snf_inject_stats::inj_pkt_send

Number of packets sent by this injection endpoint

7.2.2.2 uint64_t snf_inject_stats::nic_bytes_send

Number of raw bytes sent by Hardware Interface (see nic_bytes_recv)

7.2.2.3 uint64_t snf_inject_stats::nic_pkt_send

Number of total packets sent by Hardware Interface

7.3 snf_pkt_fragment Struct Reference

Fragment for snf_inject_send_v.

Data Fields

- const void * ptr
- uint32_t length

7.3.1 Detailed Description

Fragment for snf_inject_send_v.

7.3.2 Field Documentation

7.3.2.1 uint32_t snf_pkt_fragment::length

Number of bytes

7.3.2.2 const void* snf_pkt_fragment::ptr

Packet starting address

7.4 snf_recv_req Struct Reference

Data Fields

- void * pkt_addr

- uint32_t length
- uint64_t timestamp
- uint32_t portnum
- uint32_t length_data
- uint32_t hw_hash

7.4.1 Detailed Description

Structure to describe a packet received on a data ring.

7.4.2 Field Documentation

7.4.2.1 uint32_t snf_recv_req::hw_hash

Hash calculated by the NIC.

7.4.2.2 uint32_t snf_recv_req::length

Length of packet, does not include Ethernet CRC

7.4.2.3 uint32_t snf_recv_req::length_data

Length of packet, with alignment in receive queue

7.4.2.4 void* snf_recv_req::pkt_addr

Pointer to packet directly in data ring

7.4.2.5 uint32_t snf_recv_req::portnum

Which port number received the packet

7.4.2.6 uint64_t snf_recv_req::timestamp

64-bit timestamp in nanoseconds

7.5 snf_ring_portinfo Struct Reference

Data Fields

- snf_ring_t ring
- uintptr_t q_size
- uint32_t portcnt

- uint32_t portmask
- uintptr_t data_addr
- uintptr_t data_size

7.5.1 Detailed Description

Receive ring information

7.5.2 Field Documentation

7.5.2.1 uintptr_t snf_ring_portinfo::data_addr

Address of data ring

7.5.2.2 uintptr_t snf_ring_portinfo::data_size

Size of the data ring

7.5.2.3 uint32_t snf_ring_portinfo::portcnt

How many physical ports deliver to this receive ring

7.5.2.4 uint32_t snf_ring_portinfo::portmask

Which ports deliver to this receive ring

7.5.2.5 uintptr_t snf_ring_portinfo::q_size

Size of the data queue

7.5.2.6 snf_ring_t snf_ring_portinfo::ring

Single ring

7.6 snf_ring_qinfo Struct Reference

Data Fields

- uintptr_t q_avail
- uintptr_t q_borrowed
- uintptr_t q_free

7.6.1 Detailed Description

Queue consumption information

7.6.2 Field Documentation

7.6.2.1 uintptr_t snf_ring_qinfo::q_avail

Amount of data available not yet received (approximate)

7.6.2.2 uintptr_t snf_ring_qinfo::q_borrowed

Amount of data currently borrowed (exact)

7.6.2.3 uintptr_t snf_ring_qinfo::q_free

Amount of free space still available (approximate)

7.7 snf_ring_stats Struct Reference

Data Fields

- uint64_t nic_pkt_recv
- uint64_t nic_pkt_overflow
- uint64_t nic_pkt_bad
- uint64_t ring_pkt_recv
- uint64_t ring_pkt_overflow
- uint64_t nic_bytes_recv
- uint64_t snf_pkt_overflow
- uint64_t nic_pkt_dropped

7.7.1 Detailed Description

Structure to return statistics from a ring. The Hardware-specific counters apply to all rings as they are counted before any demultiplexing to a ring is applied.

7.7.2 Field Documentation

7.7.2.1 uint64_t snf_ring_stats::nic_bytes_recv

Number of raw bytes received by the Hardware Interface on all rings. Each Ethernet data packet includes 8 bytes of HW header, 4 bytes of CRC and the result is aligned to 16 bytes such that a minimum size 60 byte packet counts for 80 bytes.

7.7.2.2 uint64_t snf_ring_stats::nic_pkt_bad

Number of Bad CRC/PHY packets seen by Hardware Interface

7.7.2.3 uint64_t snf_ring_stats::nic_pkt_dropped

Number of packets droped, reflected in Packets Drop Filter in Counters.

7.7.2.4 uint64_t snf_ring_stats::nic_pkt_overflow

Number of packets dropped by Hardware Interface

7.7.2.5 uint64_t snf_ring_stats::nic_pkt_recv

Number of packets received by Hardware Interface

7.7.2.6 uint64_t snf_ring_stats::ring_pkt_overflow

Number of packets dropped because of insufficient space in receive ring

7.7.2.7 uint64_t snf_ring_stats::ring_pkt_recv

Number of packets received into the receive ring

7.7.2.8 uint64_t snf_ring_stats::snf_pkt_overflow

Number of packets dropped because of insufficient space in shared SNF buffering

7.8 snf_rss_mode_function Struct Reference

Data Fields

- int(∗ rss_hash_fn)(struct snf_recv_req ∗r, void ∗context, uint32_t ∗hashval)
- void ∗ rss_context

7.8.1 Detailed Description

User-defined RSS hashing function parameters. Users that provide their own callbacks can generate their own hash based on the contents of a received packet. **NOTE** This feature is available only in the SNF kernel API

7.8.2 Field Documentation

7.8.2.1 void* snf_rss_mode_function::rss_context

User context that is reflected when the user-provided rss_hash_fn is called.

7.8.2.2 int(* snf_rss_mode_function::rss_hash_fn)(struct snf_recv_req *r, void *context, uint32_t *hashval)

User-provided hash function. The callback is provided with a valid snf_recv_req structure which contains a packet as received by Sniffer. It is up to the user to inspect and parse the packet to produce a unique 32-bit hash. The implementation will map the 32-bit into one of the rings allocated in snf_open. The function must return one of three values

- **0** The packet is queued in the ring based on the 32-bit hash value that is provided, which is hashval % num_rings.
- **<0** The packet is dropped and accounted as a drop in the ring corresponding to the 32-bit hash value provided by the user.

In the example below, we replace the default hash function with a hash function that sends packets to different rings at every interval of 500 packets. This approach ignores the actual packet contents and the importance of flow affinity, we just want to spread the packet analysis to different rings and threads.

```
#define MAX_RINGS 32
#define PKT_INTERVAL 500
static uint32_t cnt[MAX_RINGS];
static uint32_t cur_ring = 0;

static int
custom_hash(struct snf_recv_req *r, void *context, uint32_t *hashval)
{
   if (++cnt[cur_ring] == PKT_INTERVAL) {
      cnt[cur_ring] = 0;
      if (++cur_ring == MAX_RINGS)
         cur_ring = 0;
   }
   // Return cur_ring as the hash value, since Sniffer will apply a
   // modulo and the corresponding ring will receive the packet when
   // calling snf_recv()
   *hash_val = cur_ring;
   return 0;
}

// At snf_open time, select a custom hash approach.
struct snf_rss_params rssp;
rssp.mode = SNF_RSS_FUNCTION;
rssp.params.rss_function.rss_hash_fn = custom_hash;
rssp.params.rss_function.rss_context = NULL; // Don't need a context

// On port 0, we will open MAX_RINGS rings, use default flags,
// select an 800 MB data ring and chose our custom hashing function.
snf_handle_t hsnf;
int rc = snf_open(0, MAX_RINGS, &rssp, 800, 0, &hsnf);
if (rc) {
   perror("Error in snf_open");
   exit(EXIT_FAILURE);
}
```

7.9 snf_rss_params Struct Reference

Data Fields

- enum snf_rss_params_mode mode
- union {
  enum snf_rss_mode_flags rss_flags
  struct snf_rss_mode_function rss_function
  } params

7.9.1 Detailed Description

When using multiple rings, users can either let Sniffer how to partition the flows of incoming packets or control the hashing using specific RSS modes. The following modes are available.

- **None:** rss_params is NULL in snf_open. When no RSS mode is explicitly specified, users let the implementation chose an RSS strategy that best matches the revision of the Myri-10G NIC. Unless a specific hashing strategy is required, this approach is best in terms of performance-portability.

- **Flag-based**: rss_params sets mode to SNF_RSS_FLAGS. This mode allows users to functionally specify which parts of a packet are significant in the RSS hashing process. A functional specification leaves enough room for the Sniffer implementation to move part or all of the hash computation between hardware, firmware and software.

- **Function-based** (kernel API only): rss_params sets mode to SNF_RSS_FUNCTION. This mode guarantees the most flexibility for the user but forces the hashing to be serialized in software (note that the current generation NICs do not necessarily take a very large performance hit compared to the two other RSS modes). This approach may be required if the flag-based approach isn't flexible enough. For example, some users may require that flow affinity be maintained according to an encapsulated TCP/IP header. See snf_rss_mode_function for more details.

7.9.2 Field Documentation

7.9.2.1 enum snf_rss_params_mode snf_rss_params::mode

RSS mode

7.9.2.2 union { ... } snf_rss_params::params

RSS parameter settings, according to the mode that is selected

7.9.2.3 enum snf_rss_mode_flags snf_rss_params::rss_flags

RSS parameters for SNF_RSS_FLAGS

7.9.2.4 struct snf_rss_mode_function snf_rss_params::rss_function

RSS params for SNF_RSS_FUNCTION

Index