Packet Handler API Implementation Agreement

Revision 1.0

Editor(s):

Rajeev Muralidhar, Intel Corporation, rajeev.d.muralidhar@intel.com

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", “MAY", and "OPTIONAL" in the remainder of this document are to be interpreted as described in the NPF Software API Conventions Implementation Agreement revision 2.0.

For additional information contact:
The Network Processing Forum, 39355 California Street,
Suite 307, Fremont, CA 94538
+1 510 608-5990 phone info@npforum.org

Table of Contents

Table of Figures

1 Revision History

Revision	Date	Reason for Changes
1.0	09/20/2003	Created Rev 1.0 of the implementation agreement by taking the NPF Packet Handler API (npf2002.240.29) and making minor editorial corrections.

2 Introduction

Network nodes often do more than simply classify and forward packets; many support applications that generate and consume network traffic of their own. Such nodes have a packet service interface to the forwarding plane that lets applications send and receive packets on the node's interfaces. Existing networking stack implementations have different mechanisms for sending and receiving packets.

The emergence of network processors and efforts at the Network Processing Forum (NPF) to standardize different layers of software that "drive" the network processor have made it critical to have an industry-wide standard mechanism and API that will allow control plane applications to be developed using standard interfaces for packet handling also. This would also foster the development of network processors that can seamlessly integrate with such control plane applications and protocol stacks.

The NPF Packet Handler (PH) API Implementation Agreement (IA) supports the exchange of packets between forwarding elements (FEs) and processes executed by controls elements (CEs) in the slow path of a system. Clients of the PH API (as defined in Section 2, following) could be a network stack in the slow path, handling exceptions that the fast path was unable to forward, or an application such as the Open Shortest Path First (OSPF) routing protocol, that maintains Routing Information Management (RTM) data.

2.1 Packet Handler API Dependencies

1. The NP Forum IPv4 Unicast Forwarding API Implementation Agreement defines the IPv4 Forwarding Information Base Handle and IPv4 Next Hop structures.
2. The (future) NP Forum IPv6 Unicast Forwarding API Implementation Agreement defines the IPv6 Forwarding Information Base Handle and IPv6 Next Hop structures.
3. The NP Forum Interface Management API Implementation Agreement defines the Interface handle and the ATM VCC Address structure.

3 Packet Handler Architecture

The PH API is a conduit for packets that pass between a Control Element (control plane processor or host processor, in short CE) and Forwarding Elements (FEs) (also referred to as Network Processing Elements, or NPE) of a system. Forwarding Elements process transit packets and local packets. Transit packets are forwarded from one external network interface to another external interface. Local packets are sourced or consumed by the system (in general the CE). Through the PH API, the CE can receive network packets addressed to the system, and it can send packets destined for other systems out on the network interfaces. It can also receive, process and retransmit packets not addressed to it, but those which a FE could not process completely. This is sometimes also referred to as the "slow path" of the forwarding operation.

The PH API is the mechanism which software clients on the CE can use to receive incoming packets from FEs, and send outgoing packets to FEs, which control the external network interfaces on which packets are received from or sent to the network. When we refer to this traffic, incoming traffic is with respect to the PH API client; similarly, outgoing traffic is also with respect to the PH API client. The packets that pass through the PH API may take this path for multiple reasons, depending on the particulars of the forwarding design. These may include, among other things:

1. Packets that are addressed specifically to an address (L2 or L3) recognized by the node controlling the interface as a termination point.
2. Packets that cannot be fully processed by an FE associated with an interface because they contain options, or are control packets that the FE does not handle.
3. Packets that are intercepted by use of a filter in the FE.
4. Packets that generate errors and need further processing in the control plane.
5. Packets whose address is not recognized.
6. Packets whose protocol is not recognized or handled by the FE.
7. Packets for which L2 address resolution is required.

Metadata consisting of additional information for packet processing accompanies each packet sent or received through the API. The PH API passes incoming packets to its receiving clients, along with *Receive metadata*, which is information supplied by the PH implementation. Sending clients pass outgoing packets to the PH implementation along with information structured as *Send metadata*.

On the incoming path, the Client application can register a *default* upcall function that receives control when a local packet is received by the PH API implementation. Optionally, a client can also register an upcall function to receive only packets that match a certain metadata pattern; it can have several of these, each receiving packets that belong to a different *flow*. In this document, a *flow* is defined as a set of packets whose metadata elements match a certain specification, which can include both fully-specified and wildcard patterns. In an implementation that supports this feature, the metadata of each packet received from the FE is compared with a series of patterns (or filters) in order to decide which client (or clients) should receive the packet.

The Packet Handler API can be used by API clients or by other Packet Handler API implementations. The Packet Handler API acts as the "upper" interface to these functions layered above it. The "bottom" interface of the PH API implementation is a proprietary one. It interfaces in a completely proprietary way to any software layers below, or to the NPE or other devices and software on the FE.

3.1 Relationship with FEs

In effect, the PH API is the recipient of information from "taps" in the FE. Several different models can be used to model the interaction between the FE and the PH API implementations. (See Figure 1.) Packets may be sent to and received from the FE at specific points in the FE or from any arbitrary point. Alternately, the PH API "taps" are collectors that multiplex and demultiplex packets to and from the FE. Example "taps" are shown by the red triangles in Figure 1.

The CE-FE configuration descriptions are not within the scope of this document. The interface between the PH API implementation and the FE is completely proprietary. Cross-licensing agreements are of course possible so that one vendor's PH API implementation can communicate with multiple vendors' FEs. These are also not in the scope of this document.

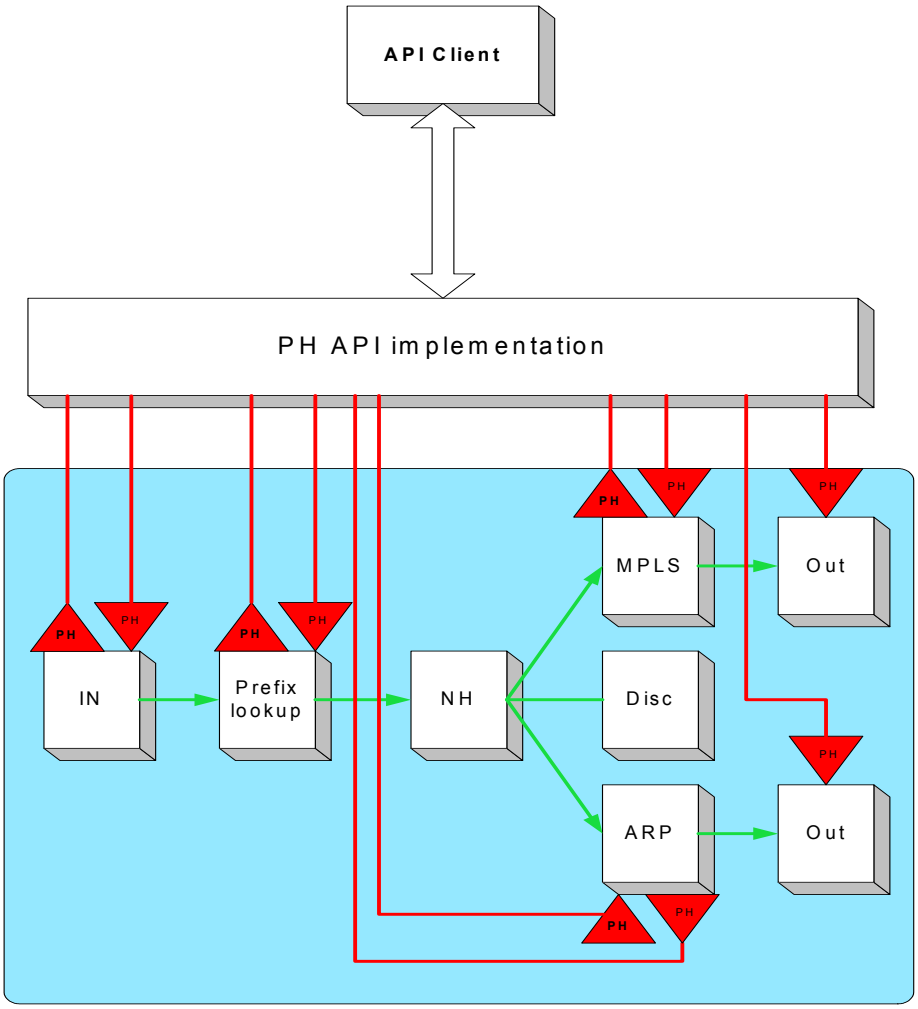

Figure 1: Communication between PH API and FE

3.2 Clients and FEs

In Figure 2 the PH APIs and the FEs are color-coded. This can be interpreted in two ways, both of them valid. First, matching PH API implementation and FE can represent matching *types*, that is, a PH API implementation written for a certain type of FE with a certain set of capabilities. Another valid interpretation is that the matching PH API implementation and FE are from the same vendor. That is, the PH API implementation was created to work with a certain vendor's FE products.

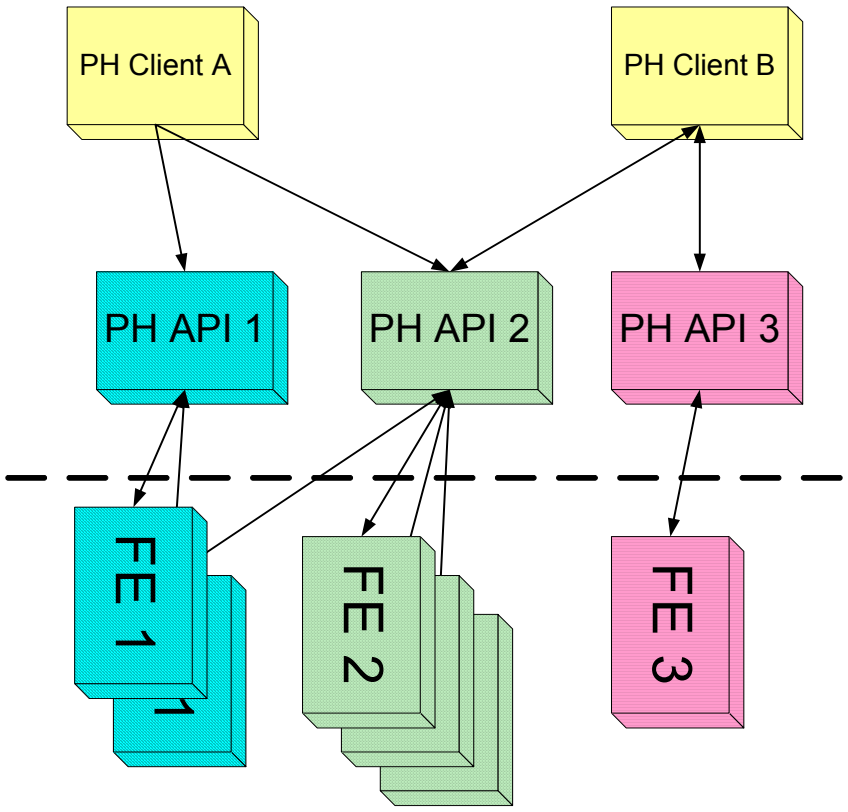

Figure 2 : Various configuration options

Figure 2 shows various options for connecting PH API clients, PH API implementations and FEs. The various options for servicing FEs are:

1. A single PH API implementation instance services only one FE instance. (PH API3 on Figure 2.)
2. A single PH API implementation instance services only one type of FE, but multiple instances of that type. (PH API1 on Figure 2.)
3. A single PH API implementation instance services multiple FE types and multiple instances of each type. (PH API2 on Figure 2.)
4. Multiple PH API implementation instances service a single FE instance. This might be the case, for example, in a single vendor implementation system where you have both active and backup instances of a PH API implementation.

Obviously many of these options imply multiple PH API instances running simultaneously in the system. This can be problematic since there are multiple instances of the same function names being exported by multiple PH API implementation instances. It is assumed that this is resolved at the system level with system-specific mechanisms. For example, in UNIX (particularly BSD-derived systems) the dlopen, dlsym, dlclose suite of functions from the standard C library can be used to access the symbols that would be duplicated by multiple instances of the PH API.

No assumption is made whether the different Packet Handler implementations run on the same processor or not. The various PH implementation instances may run on different processors, or even connect the client to the PH API implementation via some IPC mechanism, for example. In any case, the various choices are left to the system designer and are out of scope of this document.

In Figure 2, the heavy dashed line indicates the “bottom” interface between the PH API implementation and the FE(s).

On outbound traffic, the binding of the PH API implementation to one or more FEs is done through the Packet Handler metadata that is associated with a packet. For example, the binding can be made through an interface handle (see Section 4.2.1) to FE mapping table contained in the API implementation.

On inbound traffic, FEs are bound to a particular PH implementation through system specific means, either through the configuration of the FEs themselves or through some proprietary and system-specific method which directs packets to a particular implementation.

In a configuration where a single FE is serviced by more than one PH API implementation instance, the FE has to be configured so that it knows which packets must be sent to which PH API instance. Since the interfacing of the PH API implementation with the FE can be proprietary, each type of FE or vendor family of FE would have to be serviced by a corresponding PH API implementation. So, FEs must be bound to a particular PH API instance running in the system. Again, this is part of the system design and configuration and does not require additional API calls in the IA.

As shown also in Figure 2, the PH API implementation may have only one client, or it may have multiple clients. Likewise, the PH API Client may communicate with multiple PH API implementations. In order to have a system as shown in Figure 2, it is necessary for the client applications to be able to determine which packets must be sent to which PH API implementation instances.

If there are multiple clients of a PH API implementation, it is necessary to have the capability to define flows for which an upcall function for each client can be registered for packets matching different flows. It would also be possible to have multiple API clients to receive the same packets (either by not supporting flows, or by allowing multiple upcall functions to be registered for the same flow specification). Note also that in Figure 2, both Client A and Client B can access the 4 FEs through PH API number 2: The three type 2 FEs and the second type 1 FE.

3.3 Stacked PH API implementations

It is possible to stack PH API implementations. One obvious reason is to present to API Clients a single API instance with which to communicate. Note that it is not required to stack API implementations when there are multiple instances of a PH API implementation in a system (for example from different vendors). However, by stacking the PH API instances and offering an integrated (generic) PH API for client application, the task of determining which vendor- or FE-specific instance is correct for a particular packet is removed from the application. This relieves the application of the necessity of knowing which PH API instance services a particular FE or set of FEs if there are multiple FE vendor’s products in the system.

Figure 3 shows an example of a stacked configuration.

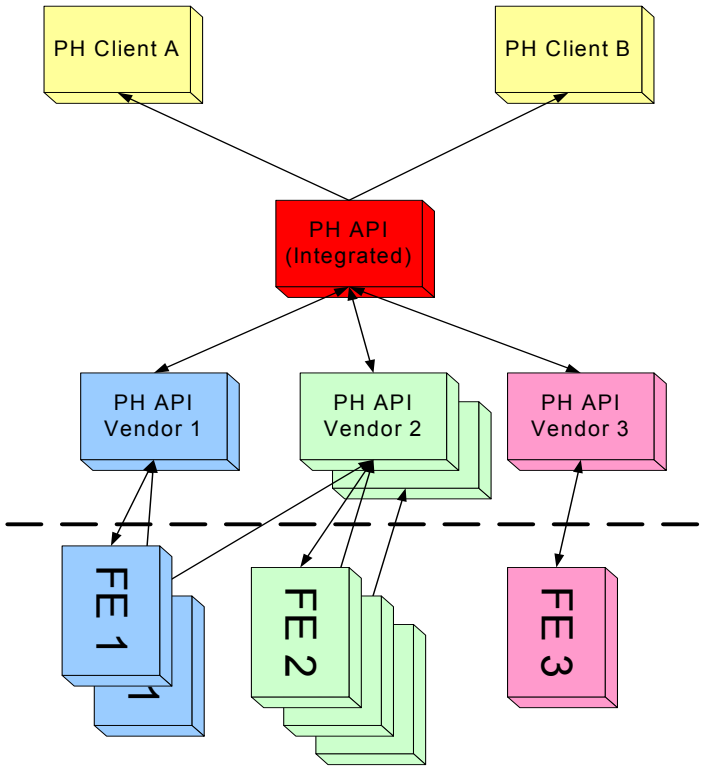

Figure 3: Stacked configuration of PH APIs

The integrated (generic) API implementation can provide some advantages for clients. It can multiplex and demultiplex traffic to/from the particular FE-specific implementations present.

It may also modify the Packet Handler metadata, for example, based on proprietary information or headers supplied by vendor-specific extensions to the PH API implementations of which it is a client (see Figure 3). Also the generic implementation could provide some non-Packet Handler functions, for example, doing interface or port lookups.

Since the different types and instances of the PH API all have the same function call names, there will be a system-dependent way to differentiate between them. The system integrator must provide a way for the application to “connect” to the integrated PH API implementation and also provide a way for the integrated PH API implementation to “connect” to the type-specific or vendor specific PH API implementations.

4 Packet Handler API Conceptual Elements

The Packet Handler API conceptual elements consist of metadata parameters that are used to identify packets and packet flows. Metadata information is used to identify key elements of a packet and/or a packet flow. Metadata information accompany the packet in both transmit and receive directions. The following subsections provide details about all the metadata information used by the PH API and the mechanisms available for PH clients for sending and receiving packets.

The remainder of this section is organized as follows: Section 4.1 begins by introducing the concept of metadata information and its relation to packets and packet flows. Subsequently, the different metadata parameters used in the send direction are described. Metadata parameters used in the receive direction are then described, followed by the notion of send and receive flows, which form the optional part of the PH API. Packet Handler Events and Priorities are then described which aid in flow control of packets between the PH clients and the PH API implementation. Finally, the different statistics counters maintained by the PH API are described.

4.1 Metadata and flows

Metadata information identifies key elements of a packet flow – for example, the protocol header type, the outgoing interface, incoming interface, etc. The metadata information is presented to the API by the PH client with the packet being sent. Correspondingly, the PH implementation delivers similar metadata along with the packets when they are being delivered into PH clients. Optionally, applications can build what are termed flow specifications using these basic data types representing metadata. Thus, a flow specification is essentially a combination of metadata information that can indicate the kind of packets being sent (send flow specification) and/or received (receive flow specification) by the application. Applications can register send and receive flow specifications with the PH API and can subsequently use the respective flow handles to identify such packet flows. Since receive flows may have wildcards, the PH API implementation must always include the metadata for all received packets that are delivered to PH clients.

The PH API flow specification is similar to a packet classification filter. Filters have been traditionally used to classify packets based on fields in packet headers or payload. The PH API implementation uses the PH flow specification to determine how and where the packet is sent or received. Specifying function calls for defining classification filters for the NPE is not in the scope of the PH API.

4.2 Metadata

The PH API provides mechanisms for specifying metadata for packets being sent or received. Applications must provide metadata for packets they are sending. The API implementation also provides metadata with the packets that it delivers to applications. The following sub sections describe the metadata in detail for both sending and receiving packets.

In general, the metadata provided by the API supports the following requirements:

1. Identify the receive interface for packets that are delivered to applications.
2. Identify the header protocol for both sending and receiving packets.
3. Direct output packets to a specific interface or next hop
4. Prioritize traffic through the API (in case there are queues in the API implementation)
5. Allow different FE processing for locally sourced packets and for transit packets

4.2.1 Controlling the path of packets

The PH API allows the application to control the path of a packet in several ways.

For outbound unicast, multicast or broadcast packets, an application can take the forwarding decision about which interface to send the packet to, and subsequently, pass the outgoing interface information along with the packet to the PH API. For outbound IP unicast packets, an application can take a forwarding decision, and subsequently pass next hop information, as defined by the "NPF IPv4 or IPv6 Unicast Forwarding API IAs" along with the packet to the PH API. In some systems, for outbound IP unicast packets, an application may let a layer below take the forwarding decision, and if more Forwarding Information Bases (FIBs) are present, pass along with the packet to the PH API, a particular FIB identifier to be used for the forwarding lookup. Note: Applications such as RSVP or RSVP-TE may insert more specific forwarding information in the packet headers.

Furthermore, optionally, on a system on which there is support for the IPv4, or IPv6 Unicast Forwarding API "discrete mode", an application may pass the "Next Hop ID", as defined by the "NPF IPv4 or IPv6 Unicast Forwarding API IAs", along with a IP unicast packet to the PH API. In such a case, passing the "Next Hop ID" instead of next hop or output interface information, allows layers below the application to forward the IP unicast packet, based on information retrieved from a Next Hop Table Entry, as specified in the NPF IPv4, or IPv6 Unicast Forwarding API IAs.

For inbound packets, the PH API implementation provides the application with information that identifies the path the packet arrived on. This information is structured in the "Logical Link Identifier".

4.2.1.1 Logical Link Identifier

Sending specification allows the PH client to specify a particular outgoing/incoming interface by means of a Logical Link Identifier.

A Logical Link Identifier refers to a physical port, physical/logical interface, and/or ATM VCC, etc. The logical link identifier is used as metadata for both sending and receiving.

4.2.2 PH API Metadata for sending packets

Applications can have different needs for sending packets, as mentioned earlier. The packets can be sent with additional information to carry out further processing requirements. Some specific examples are:

1. Layer 3 (e.g. IP) applications might have already performed some routing decision and would like to specify some “routing hints” to the API that can be used by the NPE. Additionally, L2 encapsulation needs to be added.
2. Layer 2 applications might have composed their own L2 header also. Such packets must be directed to the correct output interface/port.
3. The API does not preclude an implementation that might choose to implement higher layer protocol functionality (Layer 4 and above) so that applications can pass in packets with corresponding headers, and leave it to the implementation or the NPE to take care of the remaining headers.

Sending metadata consists of:

- Priority – To allow for API implementations to prioritize traffic through the API implementation to the underlying device. The detailed semantics of the priority metadata field is described in Section 4.2.6.
- Protocol Type – this indicates the protocol header type of the packet being passed in.
- Sending Specification – This specifies “routing hints” that applications can provide about how to send the packet out. This must be one of:

  - FIB Identifier: This is used if the application wants to indicate a particular routing table on the NPE to use for performing a lookup. It depends on the underlying NPE's capability to use this information. The FIB identifier could refer to either an IPv4 FIB or IPv6 FIB.
  - Logical Link Identifier: This could be used to indicate a specific output port to use to send the packet out on. For example, L2 applications that compose their own L2 header could use this to simply indicate the egress interface.
  - Next Hop identifier: This is an additional routing hint that could be used by the underlying device for sending the packet out.
  - Next Hop information: This consists of both a Next Hop IP Address as well as an Interface Handle. This could be either an IPv4 or IPv6 Next Hop.
- Send Flags - currently, only "Locally sourced" is supported. The purpose of this is to suppress some filtering, TTL decrement, etc.

4.2.3 PH API Metadata for receiving packets

Packets that are delivered to applications have metadata passed up with them. Applications can register a default upcall function to receive all packets. Optionally, applications can register receive flows to receive packets that match certain metadata parameters. If there are no registered applications for the received packet, the API implementation can choose to drop the packet. The mechanism for registering receive flows is described in Section 4.2.5. For all packets that are being delivered to PH clients, the PH API implementation must indicate the link on which the packet arrived on, and the PH client can infer the protocol type from the logical link identifier.

The PH API receive metadata delivered into the application consists of:

- Priority – This priority can be used in the PH API implementation to prioritize traffic between the NPE and the application. It is to be noted that this priority might be related to any kind of QoS priority/treatment the packet gets in the inbound direction although this is not specified by this IA.
- Logical Link Identifier - this identifies the logical link (interface, ATM connection, MPLS path, etc.) on which the packet arrived. If the client needs to know the protocol identity at the first byte of the packet, it must be able to infer this value from the logical link. If the NPE removes one or more headers, the logical link identifier must permit the client to identify the protocol of the header appearing first in the buffer (the first header following the ones that were removed).
- Optional offset information containing a protocol ID and offset value. These parameters are only used when the NPE has already performed some packet processing and wishes to share some additional information with the API client.
  - Offset is a number ranging from zero to length-1, where the length refers to the length of the packet. The first byte of the protocol header (as identified by the protocol ID) is located at *offset* bytes from the beginning of the packet.
  - The protocol ID of the header indicated at the specified offset. Note that if the offset is zero, protocol ID is also zero, as the protocol is inferred from the logical link identifier.
- Reason code – This indicates the reason for delivering the packet from the NPE/device to the PH API implementation. This must be one of:
  - Matched a configured filter – this indicates that the incoming packet matched a particular classification filter installed in the device.

- Destination address not found – if the underlying NPE cannot send or forward a packet because the destination address could not be found in a lookup table, the device might hand off the packet to the control plane.
- Locally addressed packet – this indicates that the packet being delivered was destined to one of the local interfaces of the device. This could be destined to some protocol/application on the control plane. For example, OSPF packets would be destined to one of the local interfaces.
- Supported protocol, unsupported option – if the device cannot handle the options being specified in the packet, it could hand it off to the control plane. For example, if the packet was an IP packet and had IP options enabled, but the device cannot process the Record Route option, it could hand the packet off to the control plane.
- Unsupported protocol – if the device cannot handle the protocol of the incoming packet, it could hand off the packet to the control plane.

- Reason sub code – this is used to provide more detailed information under certain reason codes. It is currently defined only for the 'matched a configured filter' reason code; in this case it identifies the filter rule that matched.

4.2.4 Packet Handler Send Flows

Applications can also specify the kind of packets they are sending using flow specifications. A flow specification is essentially a combination of metadata information that can indicate the kind of packets being sent and is hence called a send flow specification. Applications can thus register send flow specifications with the PH API and can subsequently use the respective flow handles to identify such packet flows. Support for send flow specification in the PH API is optional.

The send flow specification cannot have wildcards for any of its constituent metadata parameters. The metadata to be specified is similar to that to be used during sending packets as specified in Section 4.2.1.1 and are as follows:

- Priority – This will indicate the priority of the packets that belong to the flow being specified. It is to be noted that this priority might be related to any kind of QoS priority/treatment the packet gets in the outbound direction although this is not specified by this IA.
- Protocol type – this indicates the header protocol of the packet being passed in.
- Sending specification – This specifies "routing hints" that applications can provide about how to send the packet out. This could be one of:
    - FIB identifier – This indicates a particular routing table on the NPE to use for performing a lookup. It depends on the underlying NPE's capability to use this information. The FIB identifier could refer to either an IPv4 FIB or IPv6 FIB.
    - Logical Link Identifier – This indicates a specific output port to use to send the packet out on. For example, L2 applications that compose their own L2 header could use this to simply indicate the egress interface.
    - Next Hop identifier – This is an additional routing hint that could be used by the underlying device for sending the packet out.
    - Next Hop information – This consists of both a Next Hop IP Address as well as an Interface Handle. This IA specifies only information related to IPv4 and IPv6 next hops.
    - Send flags – currently, only "Locally sourced" is supported. The purpose of this is to suppress some filtering, TTL decrement, etc.

When a send flow specification is successfully registered with the PH API implementation, the PH client gets a send flow handle that must be used in subsequent calls to send the packets out. When the send flow is no longer required, the PH client should delete the send flow specification from the PH API implementation.

4.2.5 Packet Handler Receive Flows

Applications can also specify the kind of packets they expect to receive using receive flow specifications. Similar to send flow specification, a receive flow specification is essentially a combination of metadata information that can indicate the kind of packets expected by the application. Applications can register receive flow specifications with the PH API; when the PH API receives packets that match registered flows, the corresponding upcall into the PH client indicates the respective flow handles to identify the packet flows. Support for receive flow specification in the PH API is optional.

The metadata to be specified for creating a receive flow specification are as follows:

- Logical Link Identifier - this identifies the logical link (interface, ATM connection, MPLS path, etc.) on which the packet flow is expected. This can be a wildcard value to signify interest in receiving packets arriving on any logical link.
- Protocol - The protocol of the packet header. This can be a wildcard value – a value of `NPF_PROTO_FAM_ANY` indicates interest in all protocol packets.
- Reason code – This indicates the reason for delivering the packet from the NPE/device to the PH API implementation. A reason code of `NPF_PH_RECV_REASON_CODE_ANY` indicates interest in all packets that can be delivered by the forwarding plane. Other reason codes can be one of:
    - Matched a configured filter – this indicates that the incoming packet matched a particular classification filter installed in the device. As mentioned earlier, the PH API does not specify APIs to install filters and hence the application would have to use some other mechanism to install filters on the NPE.
    - Destination address not found – if the underlying NPE cannot send or forward a packet because the destination address could not be found in a lookup table, the device might hand off the packet to the control plane.
    - Locally addressed packet – this indicates that the packet being delivered was destined to one of the local interfaces of the device. This could be destined to some protocol/application on the control plane. For example, OSPF packets would be destined to one of the local interfaces.
    - Supported protocol, unsupported option – if the device cannot handle the options being specified in the packet, it could hand it off to the control plane. For example, if the packet was an IP packet and had IP options enabled, but the device cannot process the Record Route option, it could hand the packet off to the control plane.
    - Unsupported protocol – if the device cannot handle the protocol of the incoming packet, it could hand off the packet to the control plane.
- Reason sub code – this is used to provide more detailed information under certain reason codes. It is currently defined only for the ‘matched a configured filter’ reason code; in this case it identifies the filter rule that matched. A value of `NPF_PH_RECV_REASON_SUBCODE_ANY` indicates interest in all sub-codes.

4.2.6 Prioritizing traffic in the Packet Handler API

Priority is a metadata element passed with both receive and transmit packets. It can be used by an API implementation that has packet queues within it: if its queues become congested, it can relieve congestion

by discarding packets in order of priority, lowest priority first. Packets of one priority sent by one client should be delivered to the FE in the same order they were given by the client. It is to be noted that this priority might be related to any kind of QoS priority/treatment the packet gets in the inbound/outbound direction although this is not specified by this IA. There is no other use intended for this priority value.

The API defines 8 different levels of priority indicating precedence in processing packets that can be supported by the underlying implementation (values 0-7). 0 is normal precedence and 1-7 are relatively higher. Although there are eight possible priority values, various vendors may differ on the choice of the number of priority levels supported. A recommended method to merge these two divergent ranges of priority levels, i.e. eight levels in the application and two to three levels in the NPE would be to map the priority levels specified by the application to a smaller range used by the NPE. For example, in the case of a vendor that supports only 2 levels of priority queues underneath, the potential eight priority values specified by the application may be segregated by the API as shown in **Figure 4**. Irrespective of how many priority levels are provided by an implementation, it is important to have a common understanding of how the priority levels behave; the following should be implemented by PH API vendors to ensure interoperability between system vendors.

- *Priority 0 is always assured to be the lowest and 7 is assured to be the highest.*
- *Priority N packets are always given lower priority than N+1, for all N.*

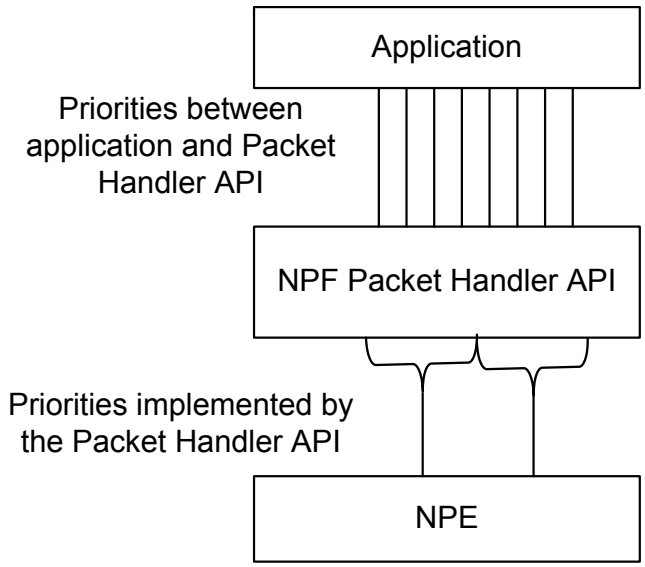

Figure 4: Mapping priorities between the application and the PH API implementation

Table 1 shows how the priorities at the API should be mapped to the priorities supported by the PH implementation. For example, if there are only 2 levels supported by the PH API implementation, P0 for low priority traffic and P1 for high priority traffic, the API maps priority levels (as used by the application) 0-3 to map to priority P0, and priority levels 4-7 to map to priority P1. Similarly the other priority levels can also be mapped to the actual number of priorities implemented.

The API implementation should maintain statistics for all priority levels that are supported as described in Section 4.5. If the API implementation does not support all priority levels, the leftmost number in the middle column of Table 1 is the index to be used for retrieving the appropriate statistics counter.

Number of priorities implemented	**Priorities at the PH API (as seen by the application)**	**Implementation Priorities (P0 is lowest)**

1	0-7	P0
2	0-3	P0
	4-7	P1
3	0-2	P0
	3-4	P1
	5-7	P2
4	0-1	P0
	2-3	P1
	4-5	P2
	6-7	P3
5	0-1	P0
	2-3	P1
	4-5	P2
	6	P3
	7	P4
6	0-1	P0
	2-3	P1
	4	P2
	5	P3
	6	P4
	7	P5
7	0-1	P0
	2	P1
	3	P2
	4	P3
	5	P4
	6	P5
	7	P6
8	0	P0
	1	P1
	2	P2
	3	P3
	4	P4
	5	P5
	6	P6
	7	P7

Table 1: Mapping NPF Packet Handler API priorities to underlying implementation

4.3 Buffer descriptor and Memory ownership

The PH API does not define the structure of the packet. The exact nature and structure of the buffer containing the packet is dependent on the operating system being used and the buffer mechanisms used by the system. The operating system might use contiguous memory or buffer chaining (like mbufs, for example). The PH API will not attempt to impose any restriction on what buffer mechanism is being used. The API simply defines a generic buffer descriptor that will be used to map to different operating system buffers. The exact manner in which this mapping is accomplished, and other details such as buffer allocation/de-allocation, buffer ownership mechanisms, etc. are out of the scope of this API definition.

The buffer descriptor contains the following information:

1. Buffer descriptor containing pointer and length.
    a. Pointer is either to the first byte of the packet, or to some structure chosen or defined by the implementer, and such structure MUST provide complete information about the packet's location in memory. Note that if this pointer refers to a structure then the structure may contain information redundant with the remainder of the parameters described here, including buffer length (1.b) and logical link identifier. In such implementations it is the responsibility of the PH implementation to ensure that the value passed in parameter 1.b and logical link identifier is consistent with any values contained within the internal structure referred to by this pointer.
    b. Length is the total length of the packet, in bytes.

4.4 Packet Handler API Events

The main purpose of events in the PH API is to allow clients to know how fast they can transmit packets without causing congestion in the PH API implementation. The PH API's send function returns an error code if the API implementation's queues and/or resources are not available. The client can then react to the backpressure and either try to retransmit at a later time, or depending on which other priorities are available, it can try to send the packet with a different priority. It can also register with the PH API implementation to receive an event to get notified when the PH API implementation resources and/or queues become available; the client can attempt to transmit packets at this time.

Packet Handler events are generated into all applications (that have registered for PH events) when resources become available after an application previously tried to transmit packets and received an error (due to some resources being unavailable in the PH API implementation). This means that the error (on transmit) might be received by one application, but multiple applications would get notified about the resource becoming available.

The API implementation will not provide any guarantee on delivering the events to all applications interested in PH events. The PH API implementation will do its best to do, and the application can implement timeout/retransmit mechanisms, or any other mechanisms to retransmit the packet.

4.5 Packet Handler API Statistics

The PH API maintains queues per priority level in both transmit and receive directions, although this IA does not specify exactly how many queues are to be supported. Correspondingly, statistics counters must be maintained by the API implementation. Counters must be differential; that is, they keep their value after a read and keep incrementing for each object countered, so that when the largest number that can be represented by the width of the counter is reached, the counter rolls over using twos-complement binary arithmetic. The counters are unsigned and 32 bits wide. The difference in value between two consecutive

reads provides the number of objects counted since the first of the two reads. The counters maintained per priority level are:

- Number of packets/frames transmitted. The number of packets or frames transmitted to the NPE.
- Number of bytes transmitted. The number of bytes, including packet headers, but excluding metadata transmitted to the NPE.
- Number of outgoing packets/frames dropped. The number of packets/frames passed into the packet handler by clients that were dropped, possibly due to congestion.
- Number of packets/frames received. The number of packets or frames received from the NPE.
- Number of bytes received. The number of bytes, including packet headers that are received from the NPE. So, for example, if there are switch fabric headers that encapsulate a frame coming into the processor board, these are excluded from the byte count because they should be stripped before the packet is handed to the PH API implementation. Here the number of bytes counted is taking into consideration the entire packet, starting from offset 0.
- Number of incoming packets/frames dropped. The number of packets/frames passed into the packet handler implementation from the NPE, but were dropped, possibly due to congestion.
- Number of incoming packets dropped by the PH API implementation. The number of incoming packets dropped because they did not belong to any registered flow and there was no default receive function.

As mentioned in Section 4.2.6, the API implementation should maintain statistics for all priority levels that are supported. If the API implementation does not support all priority levels, the leftmost number in the middle column of Table 1 is the index to be used for retrieving the appropriate statistics counter.

5 Packet Handler API Data Types

5.1 Common

5.1.1 Callback Handle

```
typedef NPF_uint32_t NPF_callbackHandle_t;
```

This is the callback handle defined in the NPF Software Conventions document.

5.2 Buffer Descriptor

```
/**********************************
NPF Buffer descriptor structure.

1. Buffer descriptor containing pointer and length.
(a) Pointer is either to the first byte of the packet, or to some
structure chosen or defined by the implementer.
(b) Length is the total length of the packet, in bytes.
**********************************/
typedef struct {
     NPF_uint8_t *packetStart;
/* pointer to first byte of packet OR some structure containing
the first byte of the packet */
     NPF_uint32_t packetLen;   /* payload length */
} NPF_phBufDescr_t;
```

5.3 Metadata

5.3.1 Protocol Type

`NPF_protocol_t` is a structure that identifies the protocol type of a header. `NPF_protocol_t` consists of two variables: a protocol family identifier, and a protocol number. The following protocol families are defined:

- LLC LSAP identifiers
- Ethernet Type numbers
- PPP Numbers
- IP Protocol numbers
- NP Forum defined protocol numbers

The actual protocol numbers for each family will be the same as those defined by IANA and other standards bodies. For convenience, IANA maintains a listing of all these numbers, even though it might not be in their domain. The protocol numbers of protocols belonging to the families above are available at:

LLC LSAP protocol numbers: http://www.iana.org/assignments/ieee-802-numbers

Ethernet protocol numbers: http://www.iana.org/assignments/ethernet-numbers. Similar information for Ethernet numbers may also be found at http://standards.ieee.org/regauth/ethertype/type-pub.html.

PPP numbers: http://www.iana.org/assignments/ppp-numbers

IP protocol numbers: http://www.iana.org/assignments/protocol-numbers

```
/**********************************
Protocol Type Indicator

This structure identifies a protocol by its protocol number.
Protocol numbers are assigned by several different numbering
authorities, so there is a protocol family identifier to say
what number series the protocol number belongs to.  The IEEE and
IETF number assignments can be found on the IANA web site,
http://www.iana.org/numbers.html.
**********************************/
typedef enum {
     NPF_PROTO_FAM_LLC = 1,/* IEEE 802 LSAP assignments */
     NPF_PROTO_FAM_ET = 2,/* IEEE EtherType assignments */
     NPF_PROTO_FAM_PPP = 3,/* IETF PPP protocol numbers */
     NPF_PROTO_FAM_IP = 4, /* IETF IP protocol numbers */
     NPF_PROTO_FAM_NPF = 5,   /* NP Forum-defined numbers */
     NPF_PROTO_FAM_UNK = 6,   /* Unknown protocol */
     NPF_PROTO_FAM_ANY = 7    /* Wildcard value */
} NPF_protocolFamily_t;

typedef struct {
     NPF_protocolFamily_t     family;
     NPF_uint16_t             protocol;
} NPF_protocol_t;
```

5.3.2 Priority

As described earlier, it can be used by a PH implementation that has packet queues within it: if its queues become congested, it can relieve congestion by discarding packets in order of priority, lowest priority first. This priority does not indicate priority of transmission out of the NPE – it only indicates the priority of service outbound from the PH implementation.

```
/**********************************
Priority

Priority is a metadata element passed with both receive and
transmit packets.  It can be used by an API implementation that
has packet queues within it: if its queues become congested, it
can relieve congestion by discarding packets in order of
priority, lowest priority first.  There is no other use intended
for this priority value.
**********************************/
```

```
typedef NPF_uint8_t NPF_phPriority_t;

#define NPF_PH_PRIORITY_LOWEST 0
#define NPF_PH_PRIORITY_HIGHEST 7
```

5.3.3 Send Flags

The Send Flags metadata element contains flags that can condition the operation of the NPE for sending specific kinds of packets.

```
/**********************************
Send Flags

The Send Flags metadata element contains boolean flags that can condition the operation of the NPE for sending specific kinds of packets.
**********************************/
typedef NPF_uint32_t NPF_phSendFlags_t;

#define NPF_PH_SEND_FLAGS_LOCAL_SOURCE 0x00000001

/* Locally sourced indicates that the packet originates from the local host system, and should be forwarded as such.  This can be used by the PH implementation and the FE together to make sure the TTL is not decremented by the FE, that input firewall filters are bypassed, etc., as appropriate to the implementation.
*/
```

5.3.4 Receive Reason and Sub Code

```
/**********************************
Reason Code and Sub Code
**********************************/
typedef enum
{
      NPF_PH_RECV_REASON_CODE_ANY = 0,
      NPF_PH_RECV_REASON_CODE_NO_DEST_ADDR = 1,
            /*destination address was not found */
      NPF_PH_RECV_REASON_CODE_LOCALLY_ADDRESSED = 2,
            /* locally destined packet */
      NPF_PH_RECV_REASON_CODE_UNSUPPORTED_OPTION = 3,
            /* Unsupported option */
      NPF_PH_RECV_REASON_CODE_UNSUPPORTED_PROTOCOL = 4,
            /* Unknown protocol */
      NPF_PH_RECV_REASON_CODE_FILTER_MATCH = 5
            /* Matched a filter */
} NPF_phRecvReasonCode_t;
```

```
typedef NPF_uint32_t NPF_phRecvReasonSubcode_t;

#define NPF_PH_RECV_REASON_SUBCODE_ANY 0
```

5.3.5 Logical Link Identifier

```
/**********************************
Logical Link Identifier Structure

A logical link is like an interface, but more specific.  It
identifies an input or output link as a LAN port, a point-to-
point interface like POS, or, for subdivided interfaces like
ATM, it indicates a specific VPI/VCI.  It can be extended to
indicate a specific MPLS path as well.
**********************************/
typedef struct
{
     NPF_IfHandle_t  interfaceHandle;
     NPF_IfType_t   intfType;
     union
     {
         NPF_uint32_t  unused;
         NPF_VccAddr_t vpiVci;
     } u;
} NPF_LogicalLinkID_t;

#define NPF_INTERFACE_HANDLE_ANY   0
```

The above definition of `NPF_INTERFACE_HANDLE_ANY` depends on the NPF Interface Management implementation. Specifically, an Interface handle of zero is implicitly meant to be null or invalid (`NPF_IF_E_INVALID_HANDLE`) in the NPF Interface Management implementation.

The logical link identifier is used to translate to a physical port on the forwarding plane. The interface handle field could be used to indicate a wildcard. In the case of an ATM interface, for example, the application will need to indicate the ATM VPI/VCI. This structure could be further extended to include support for multicast and other kinds of interfaces like Frame Relay, etc.

5.3.6 Sending Specification

These definitions are from the NPF IPv4 Unicast API, the NPF Interface API Implementation Agreement and NPF IPv6 Unicast API. The FIB handle is a generic handle representing a FIB (Forwarding Information Base) on the NPE. The output interface could also be used to specify routing hints (for example, for multicast packets, L2 packets, etc.). The output specification could also be the same information that is passed into the NPF IPv4 API as described in Section 4.2.1.

```
/* FIB handles from NPF IPv4 API and NPF IPv6 API*/
typedef NPF_uint32_t NPF_IPv4UC_FwdTableHandle_t;
typedef NPF_uint32_t NPF_IPv6UC_FwdTableHandle_t;

/**********************************
```

```
Sending Specification

The Sending Specification is metadata that accompanies output
packets. In various ways, it can convey information to the FE
about how a forwarding decision should be made for this packet.
The possibilities are (mutually exclusive):
  - No specification given
  -  Send the packet out a specific logical link (i.e.,
interface,
    ATM connection, MPLS path, etc.)
  - Use a particular forwarding table in a case where the FE
    does a destination address lookup to forward the packet
  -  Use a particular next hop (interface and next hop IP
address).
It can be extended to include other protocols that may need to
be supported.
***********************************/
typedef enum
{
     NPF_PH_SEND_SPEC_NONE = 1,
     NPF_PH_SEND_SPEC_TYPE_OUTPUT_LINK_IDENTIFIER = 2,
     NPF_PH_SEND_SPEC_TYPE_IPv4FIB_IDENTIFIER = 3,
     NPF_PH_SEND_SPEC_TYPE_IPv4NEXTHOP_IDENTIFIER = 4,
     NPF_PH_SEND_SPEC_TYPE_IPv4NEXTHOP_INFO = 5,
     NPF_PH_SEND_SPEC_TYPE_IPv6FIB_IDENTIFIER = 6,
     NPF_PH_SEND_SPEC_TYPE_IPv6NEXTHOP_IDENTIFIER = 7,
     NPF_PH_SEND_SPEC_TYPE_IPv6NEXTHOP_INFO = 8
} NPF_phSendSpecType_t;

typedef struct
{
     NPF_IPv4Address_t nextHopAddr;
     NPF_IfHandle_t   interfaceHandle;
} NPF_IPv4NextHopInfo_t;

typedef struct
{
     NPF_IPv6Address_t   nextHopAddr;
     NPF_IfHandle_t      interfaceHandle;
} NPF_IPv6NextHopInfo_t;

typedef struct
{
     NPF_phSendSpecType_t       SendSpecType;
     union
     {
         NPF_uint32_t                   unused;
```

```
        NPF_LogicalLinkID_t             logicalLink;
        NPF_IPv4UC_FwdTableHandle_t     v4fibHandle;
        NPF_IPv6UC_FwdTableHandle_t     v6fibHandle;
        NPF_uint32_t                    nextHopIdentifier;
        NPF_IPv4NextHopInfo_t           v4nextHop;
        NPF_IPv6NextHopInfo_t           v6nextHop;
    } u;
} NPF_phSendSpec_t;
```

This Implementation Agreement requires that all API implementations must provide support for all the above sending specifications.

5.3.7 Packet Handler Send Metadata

This structure contains all the metadata needed for Lsending a packet. It is used when sending a packet using the `NPF_PHPacketSendTo()` function. It contains:

- Protocol - this is the header protocol type
- Priority - Priority of the packet (if supported by the implementation)
- Sending Specification - Routing hints for sending the packet out.
- Send Flags - Locally sourced, etc.

```
/*************************************************
Send Metadata
*************************************************/
typedef struct
{
    NPF_protocol_t      protocolType;
    NPF_phPriority_t    priority;
    NPF_phSendSpec_t    sendSpec;
    NPF_phSendFlags_t   sendFlags;
} NPF_phSendMetadata_t;
```

5.3.8 Packet Handler Receive Metadata

Receive Metadata is the structure passed up to the client with a received packet. It contains the following:

- Priority – priority that the packet was treated with in the PH API implementation
- Reason code and sub code – indicates why the packet is being delivered to the PH client
- Logical Link Identifier – This identifies the logical link (interface, ATM connection, MPLS path, etc.) on which the packet arrived. If the client needs to know the protocol identity at the first byte of the packet, it must be able to infer this value from the logical link.
- Offset information - contains a protocol ID and offset value. These parameters are only used when the NPE has performed some packet processing already and wishes to share with the API client some additional information. Offset is a number ranging from zero to length-1. At the offset (bytes) from the beginning the packet is the first byte of a protocol header identified by the Protocol ID.

The protocol ID of the header indicated at the offset. Note that if offset is zero, this value is also zero, as the protocol is inferred from the logical link identifier.

```
/**********************************
Receive Metadata
**********************************/

typedef struct
{
     NPF_uint32_t              offset;
     NPF_protocol_t            protocolID;
     NPF_LogicalLinkID_t       logicalLink;
     NPF_phRecvReasonCode_t    reasonCode;
     NPF_phRecvReasonSubcode_t reasonSubCode;
     NPF_phPriority_t          priority;
} NPF_phRecvMetadata_t;
```

5.3.9 Packet Handler Send Flow

```
/**************************************************
Send Flow Specification

This structure contains all the metadata needed for sending a
packet. It is used when creating a send flow.

It has the same contents as the Send Metadata
**************************************************/
typedef NPF_phSendMetadata_t NPF_phSendFlow_t;
```

5.3.10 Send Flow Handle

The PH API client supplies sending metadata when it sends packets via the Packet Handler. If it calls the `NPF_PHPacketSend()` function, it supplies metadata each time it sends a packet. As an optimization, the API defines the concept of a Send Flow – a shorthand for metadata that is the same for every packet in the flow. The client supplies the sending metadata when it creates a flow. The API implementation stores the metadata and returns a flow handle, which the client can use when sending packets using the `NPF_PHPacketSend()` function.

```
/**********************************
Send Flow Handle
**********************************/
typedef NPF_uint32_t NPF_phSendFlowHandle_t;
```

5.3.11 Packet Handler Receive Flows

Applications register interest in packet flows by specifying receive flows. The API implementation returns a Recv Flow handle that will be used subsequently when the API implementation hands up packets to the application. The contents are as follows:

- Protocol Type -- matches the protocol type of the first header in the packet, as passed up from the FE. A value of NPF_PROTO_FAM_ANY in the protocol family variable matches ANY protocol.
- Logical Link -- Specifies the logical link on which the packet was received. If the Interface Handle part of this is null, it matches any logical link (i.e. packets that arrive on all/any link).
- Reason Code -- Specifies the reason code to match. A value of NPF_PH_RECV_REASON_CODE_ANY matches any reason code.
- Reason Subcode -- specifies the reason subcode to match, if the given reason code uses a subcode. NPF_PH_RECV_REASON_SUBCODE_ANY is a wildcard value.

```
/**********************************
Receive Flow Specification
***********************************************************/
typedef struct
{
     NPF_protocol_t        protocolType;
     NPF_LogicalLinkID_t   logicalLink;
     NPF_phRecvReasonCode_t    reasonCode;
     NPF_phRecvReasonSubcode_t reasonSubCode;
} NPF_phRecvFlow_t;
```

5.3.12 Receive Flow Handle

A receive flow lets a client request delivery of packets that arrive from the FE and that match certain metadata specifications. When a client registers a receive flow, the API implementation returns a handle for it.

```
/**********************************
Receive Flow Handle
***********************************/

typedef NPF_uint32_t NPF_phRecvFlowHandle_t;
```

5.3.13 PH API Statistics

Each priority that is supported has 8 counters. Each item counted has a counter for each of the eight priorities. In addition, an additional counter is for packets received that were dropped if there were no flows registered for them.

```
/**********************************************
Packet Handler Statistics
***********************************************/
typedef struct
{
     NPF_uint32_t npfPHPktsTx[8];
     NPF_uint32_t npfPHBytesTx[8];
     NPF_uint32_t npfPHTxPktsDropped[8];
     NPF_uint32_t npfPHPktsRx[8];
     NPF_uint32_t npfPHBytesRx[8];
     NPF_uint32_t npfPHRxPktsDropped[8];
```

```
     NPF_uint32_t npfPHRcvPktsDroppedNoFlow;
} NPF_phStatistics_t;
```

5.4 Data Structures for Completion Callbacks

```
typedef NPF_uint32_t NPF_phCallbackType_t;

typedef enum
{
     NPF_PH_CB_TYPE_SEND_PACKET = 1,
     NPF_PH_CB_TYPE_CREATE_SEND_FLOW = 2,
     NPF_PH_CB_TYPE_SEND_PACKET_TO = 3,
     NPF_PH_CB_TYPE_DEL_SEND_FLOW = 4,
     NPF_PH_CB_TYPE_REG_RCV_FLOW = 5,
     NPF_PH_CB_TYPE_GET_NUM_PRIORITIES = 6,
     NPF_PH_CB_TYPE_GET_STATISTICS = 7
} NPF_phCallbackType_t;

typedef NPF_uint32_t NPF_phErrorType_t;
typedef NPF_uint32_t NPF_phErrorCode_t;

/* Error codes and return values */
#define NPF_PH_BASE_ERR  (NPF_INTERFACES_MAX_ERR + 1)
#define NPF_PH_MAX_ERR (NPF_PH_BASE_ERR + 99)

#define NPF_PH_E_BAD_CALLBACK_FUNCTION  (NPF_PH_BASE_ERR+1)
#define NPF_PH_E_CALLBACK_ALREADY_REGISTERED (NPF_PH_BASE_ERR+2)
#define NPF_PH_E_BAD_CALLBACK_HANDLE (NPF_PH_BASE_ERR+3)
#define NPF_PH_E_UNKNOWN_PROTOCOL (NPF_PH_BASE_ERR+4)
#define NPF_PH_E_UNSUPPORTED_OPTION (NPF_PH_BASE_ERR+5)
#define NPF_PH_E_BAD_FIB_ID (NPF_PH_BASE_ERR+6)
#define NPF_PH_E_BAD_NEXTHOP_IDENTIFIER (NPF_PH_BASE_ERR+7)
#define NPF_PH_E_BAD_LOGICAL_LINK_IDENTIFIER (NPF_PH_BASE_ERR+8)
#define NPF_PH_E_UNSUPPORTED_FLAGS (NPF_PH_BASE_ERR+9)
#define NPF_PH_E_INVALID_SEND_FLOW_HANDLE (NPF_PH_BASE_ERR+10)
#define NPF_PH_E_INVALID_PRIORITY (NPF_PH_BASE_ERR+11)
#define NPF_PH_E_BAD_BUFFER (NPF_PH_BASE_ERR+12)
#define NPF_PH_E_TX_RESOURCE_UNAVAILABLE (NPF_PH_BASE_ERR+13)
#define NPF_PH_E_PARAMETER_NOT_SUPPORTED (NPF_PH_BASE_ERR+14)

/* Error codes from sending packet out of the device
   These errors can be reported only if the device informs
   the PH API implementation of these errors during transmission
*/
#define NPF_PH_E_SEND_FAIL (NPF_PH_BASE_ERR+15)
#define NPF_PH_E_ARP_FAIL (NPF_PH_BASE_ERR+16)
#define NPF_PH_E_INVALID_PORT (NPF_PH_BASE_ERR+17)
```

```
#define NPF_PH_E_PHYSICAL_ERROR (NPF_PH_BASE_ERR+18)
#define NPF_PH_E_INTERNAL_ERROR (NPF_PH_BASE_ERR+19)

/*********************************************
Callback Structure
This is delivered into the application for every API call it
makes. The Callback type can be used by the application to
demultiplex the response. The union contains the corresponding
response data
*********************************************/
typedef struct
{
     NPF_phCallbackType_t     type;
     NPF_phErrorType_t   error;
     union
     {
          NPF_uint32_t              unused;
          NPF_phBufDescr_t          *bufDescr;
          NPF_uint32_t         phNumPriorities;
          NPF_phStatistics_t        *phStatistics;
          NPF_phSendFlowHandle_t    phSendFlowHandle;
          NPF_phRecvFlowHandle_t    phRecvFlowHandle;
     } u;
} NPF_phCallbackData_t;
```

The following table shows the callback data that is associated with the asynchronous responses of the respective PH API functions.

Asynchronous Function	Callback Type	Callback Data
NPF_PHPacketSend	NPF_PH_CB_TYPE_SEND_PACKET	NPF_phBufDesrcr_t
NPF_PHPacketSendTo	NPF_PH_CB_TYPE_SEND_PACKET_TO	NPF_phBufDesrcr_t
NPF_PHSendFlowCreate	NPF_PH_CB_TYPE_CREATE_SEND_FLOW	NPF_phSendFlowHandle_t
NPF_PHSendFlowDelete	NPF_PH_CB_TYPE_DEL_SEND_FLOW	None
NPF_PHRecvFlowRegister	NPF_PH_CB_TYPE_REG_RECV_FLOW	NPF_phRecvFlowHandle_t
NPF_PHNumPrioritiesGet	NPF_PH_CB_TYPE_GET_NUM_PRIORITIES	phNumPriorities
NPF_PHStatisticsGet	NPF_PH_CB_TYPE_GET_STATISTICS	NPF_phStatistics_t

5.5 Data Structures for Event Callbacks

```
/*
     Event types
*/
typedef enum {
     NPF_PH_TX_RESOURCE_AVAILABLE = 0,
```

```
} NPF_phEvent_t;

/*
     Data structures for event callback
*/

/* This structure reports the minimum priority level available
for a PH API client for transmission. An array of this structure
can be reported to the client.
*/
typedef struct {
     NPF_phEvent_t  phEventType; /* PH Event */
     union {
          NPF_uint16_t   minPriority; /* Minimum priority
                    level available for transmission */
     } u;
} NPF_phEventData_t;

typedef struct {
     NPF_uint16_t   n_data; /* Number of events in array */
     NPF_phEventData_t *eventData; /* Array of events */
} NPF_phEventArray_t;
```

5.6 Callbacks

5.6.1 API Callback

Syntax

```
typedef void (*NPF_phCallbackFunc_t)(
     NPF_IN    NPF_userContext_t        userContext,
     NPF_IN    NPF_correlator_t         phCorrelator,
     NPF_IN    NPF_phCallbackData_t     *phCallbackdata
);
```

Description of function

The application registers this callback function with the NPF PH API. The callback function is implemented by the application, and is registered with the API through the `NPF_PHRegister()` function.

Input Parameters

- `userContext:` The context item that was supplied by the application when the completion callback function was registered.
- `phCorrelator:` The correlator item that was supplied by the application when an API function call was made. The correlator is used by the application mainly to distinguish between multiple invocations of the same function.
- `phCallbackData:` Pointer to a structure containing response information related to the API function call. See `NPF_phCallbackData_t` for more details.

Output Parameters

None

Return Codes

None

5.6.2 Event Callback Function

Syntax

```
typedef void (*NPF_phEventCallFunc_t) (
     NPF_IN NPF_userContext_t      userContext,
     NPF_IN NPF_phEventArray_t     phEventArray);
```

Description of function

This is the callback function (handler function) registered by a PH API client for receiving Packet Handler events. One or more events can be notified to the application through a single invocation of this event handler function. Packet Handler events are delivered to a PH API client when one or more of the priority queues are available for packet transmission. Information on which priority levels are available is represented in an array in the `phEventArray` structure.

Currently, the event NPF_PH_TX_RESOURCE_AVAILABLE is defined. This indicates to the PH client that the API implementation can now accept packets for transmission. The API does not impose any restrictions on the number of resources and/or queues that have to be present in an implementation, or on how the priority handling is to be implemented.

Input Parameters

- `userContext:` User context provided during callback function registration.
- `phEventArray:` The event array being delivered to the PH client. This is defined and implemented in accordance with the Event Handling mechanisms described in the NPF Software API Conventions Implementation Agreement.

Output Parameters

None.

Return Codes

None.

6 Packet Handler API Function Calls

6.1 Completion Callback Registration Function

Syntax

```
NPF_error_t NPF_PHRegister(
      NPF_IN     NPF_userContext_t          userContext,
      NPF_IN     NPF_phCallbackFunc_t       phCallbackFunc,
      NPF_OUT    NPF_callbackHandle_t       *phCallbackHandle
);
```

Description of function

This function is used by an application to register its completion callback function for receiving asynchronous responses related to API function calls. The application may register multiple callback functions; each callback function is identified by the tuple {`userContext`, `phCallbackFunc`}, and for each individual pair, a unique `phCallbackHandle` will be assigned by the API implementation. Subsequent calls to any of the API functions have to use the callback handle `phCallbackHandle` returned here.

It is to be noted that `NPF_PHRegister()` is a synchronous function and has no callback function associated with it.

Input Parameters

- `userContext`: A context item for uniquely identifying the context of the application registering the callback function. This is returned back to the application through the callback function when it is invoked. Applications can assign any value to the userContext and the value is completely opaque to the API implementation.
- `phCallbackFunc`: Pointer to the callback function to be registered.

Output Parameters

- `phCallbackHandle:` A unique identifier assigned for the registered userContext and phCallbackFunc pair. This handle will be used by the application to specify which callback is to be called when invoking asynchronous API functions. This will also be used when deregistering the userContext and phCallbackFunc pair.

Return Codes

- `NPF_NO_ERROR:` The registration was successful.
- `NPF_PH_E_BAD_CALLBACK_FUNC:` The callback function passed in was NULL.
- `NPF_PH_E_CALLBACK_ALREADY_REGISTERED:` No new registration was made since the userContext and phCallbackFunc pair was already registered. Whether this is treated as an error or not is dependent on the application.

6.2 Completion Callback De-registration Function

Syntax

```
NPF_error_t NPF_PHDeregister(
      NPF_IN     NPF_callbackHandle_t phCallbackHandle
      );
```

Description of function

This function is used by an application to deregister its completion callback function for receiving asynchronous responses related to API function calls. After this function returns successfully, the application cannot make any subsequent API invocations. It is to be noted that `NPF_PHDeregister()` is a synchronous function and has no callback function associated with it.

Input Parameters

- `phCallbackHandle:` The unique identifier representing the pair of user context and callback function to be deregistered. This identifier was obtained during a previous call to NPF_PHRegister().

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The de-registration was successful.
- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle. There is no effect to the registered callback functions.

6.3 Event Callback Registration Function

Syntax

```
NPF_error_t NPF_PHEventRegister(
      NPF_IN     NPF_userContext_t          userContext,
      NPF_IN     NPF_phEventCallFunc_t      phEventCallFunc,
      NPF_OUT    NPF_callHandle_t           *phEventHandle
);
```

Description of function

This function is used by an application to register its callback function for receiving asynchronous events related to PH API. It is to be noted that `NPF_PHEventRegister()` is a synchronous function and has no callback function associated with it.

Input Parameters

- userContext: A context item for uniquely identifying the context of the client registering the event handler function. This is returned back to the client through the event callback function when it is invoked.

`phEventCallFunc:`
Pointer to the event callback function to be registered.

Output Parameters

- `phEventHandle:` A unique identifier assigned for the registered userContext and phEventCallFunc pair. This handle will be used by the client to specify which callback is to be called when invoking asynchronous API functions. This will also be used when deregistering the event handler function.

Return Codes

- `NPF_NO_ERROR:` The registration was successful.
- `NPF_PH_E_BAD_HANDLER_FUNC:` The event handler function passed in was NULL.
- `NPF_PH_E_CALLBACK_ALREADY_REGISTERED:` The callback function passed in was previously registered.

6.4 Event Callback Deregistration Function

Syntax

```
NPF_error_t NPF_PHEventDeregister(
      NPF_IN     NPF_callHandle_t phEventHandle
      );
```

Description of function

This function is used by an application to deregister its event handler function. After this function returns successfully, the application will not get any PH API events. It is to be noted that `NPF_PHEventDeregister()` is a synchronous function and has no callback function associated with it.

Input Parameters

- `phEventHandle:` The unique identifier representing the pair of user context and event handler function to be deregistered. This identifier was obtained during a previous call to NPF_PHEventRegister().

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The de-registration was successful.
- `NPF_PH_E_BAD_HANDLE:` The API implementation does not recognize the event handle. There is no effect to the registered callback functions.

6.5 Function to Send a Packet

Syntax

```
NPF_error_t NPF_PHPacketSendTo(
      NPF_IN     NPF_phCallbackHandle_t   phCallbackHandle,
      NPF_IN     NPF_correlator_t         correlator,
      NPF_IN     NPF_errorReporting_t     phErrorReporting,
      NPF_IN     NPF_phBufDescr_t         *packet,
      NPF_IN     NPF_phSendMetadata_t     *phSendMetadata
);
```

Description of function

This function is used by an application to send a packet by specifying metadata. The application need not create a send flow handle in order to use this function.

The PH API implementation cannot guarantee that the packet will be transmitted. This is due to the fact that an error could occur at any stage of sending, for example, L2 address resolution, physical error, etc. Hence, a successful return from this function only implies that the packet was received successfully and the API implementation will attempt transmission.

Input Parameters

- `phCallbackHandle:` Unique callback handle obtained during callback function registration.
- `Correlator:` The application's correlator for this call. It is used to distinguish between multiple invocations of the same API function call.
- `phErrorReporting:` This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.
- `Packet:` This is a pointer to the packet to be transmitted.
- `phSendMetadata:` Pointer to a Packet Handler Send metadata.

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The packet was accepted successfully for transmission.
- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle.
- `NPF_PH_E_TX_RESOURCE_UNAVAILABLE :` The PH API implementation did not have enough resources (space in the queues for the priority, etc.) for transmitting the packet.
- `NPF_PH_E_INVALID_PRIORITY:` The priority indicated in `NPF_phSendMetadata_t` is unsupported.
- `NPF_PH_E_BAD_FIB_HANDLE:` The FIB handle indicated in `NPF_phSendMetadata_t` is not a valid FIB handle.
- `NPF_PH_E_BAD_NEXTHOP_IDENTIFIER:` The next hop identifier indicated in `NPF_phSendMetadata_t` is invalid.
- `NPF_PH_E_BAD_LOGICAL_LINK_ID:` The Logical Link identifier indicated in `NPF_phSendMetadata_t` is not valid.

- `NPF_PH_E_UNSUPPORTED_FLAGS`: The send flags indicated in `NPF_phSendMetadata_t` are not valid.
- `NPF_PH_E_PARAMETER_NOT_SUPPORTED`: The PH API implementation does not support one of the parameters being passed in. This could happen, for instance, if the Next Hop information being passed in is different from what is supported by the NPF IPv4 Unicast API implementation running on the system.

Asynchronous Response

The asynchronous response will contain the error code representing status of the call. A pointer to the buffer descriptor passed in will be passed to the callback function.

6.6 Receive Packet Function (upcall)

Syntax

```
typedef void (*NPF_PHRecvPacketFunc_t) (
     NPF_IN     NPF_callbackHandle_t   phRcvCallbackHandle,
     NPF_IN     NPF_phBufDescr_t       *packet,
     NPF_IN     NPF_phRecvMetadata_t   *phRecvMetadata
);
```

Description of function

This is the upcall function that will be invoked by the PH API implementation when it receives a packet that matches the registered flow from the forwarding plane. The API implementation also passes up metadata information about the packet that is being delivered. This is the default upcall function – that is, a PH client that has registered this upcall will receive all packets that reach the PH implementation. PH clients must register receive flows (see Section 6.14) if the PH implementation is expected to de multiplex packets into PH clients based on metadata.

It is to be noted that that receive flows might intersect between applications (or between flows registered by the same application); if so, the API implementation would have to de-multiplex the packet into all registered consumers, possibly by making copies of the packet.

Input Parameters

- `phRcvCallbackHandle`: The callback returned to the PH client previously during registration of the upcall function.
- `Packet`: Pointer to the packet that matched the receive flow.
- `phRecvMetadata`: Pointer to the receive metadata corresponding to the packet that is being delivered into the PH client.

Output Parameters

None.

Return Codes

None.

6.7 Function to Register a Receive Packet Upcall

Syntax

```
NPF_error_t NPF_PHRecvPacketRegister(
      NPF_IN NPF_userContext_t         userContext,
      NPF_IN NPF_PHRecvPacketFunc_t    phRcvPktFunc,
      NPF_OUT NPF_callbackHandle_t     *phRcvCallbackHandle
);
```

Description of function

This function is used by an application to register a default upcall function. The upcall function specified will be invoked by the API implementation to deliver packets into the PH client.

Input Parameters

- `userContext`: A context item for uniquely identifying the context of the application registering the callback function. This is returned back to the application through the callback function when it is invoked. Applications can assign any value to the userContext and the value is completely opaque to the API implementation.
- `phRecvPktFunc`: Pointer to the upcall function that will be invoked when the PH API delivers packets into the application.

Output Parameters

- `phRcvCallbackHandle`: A unique identifier assigned for the registered userContext and phRcvPktFunc pair. This handle will be delivered into the PH client when packets are delivered through this upcall. The handle will also be used when deregistering the upcall.

Return Codes

- `NPF_NO_ERROR`: The registration was successful.
- `NPF_PH_E_BAD_CALLBACK_FUNC`: The callback function passed in was NULL.
- `NPF_PH_E_CALLBACK_ALREADY_REGISTERED`: No new registration was made since the receive upcall was already registered. Whether this is treated as an error or not is dependent on the application.

6.8 Function to Deregister a Receive Packet Upcall

Syntax

```
NPF_error_t NPF_PHRecvPacketDeregister(
      NPF_IN NPF_callbackHandle_t      phRcvCallbackHandle
);
```

Description of function

This function is used by an application to deregister its receive upcall function for packet reception. After this function returns successfully, the application cannot receive any packets.

Input Parameters

- `phRcvCallbackHandle`: The callback handle received when the receive upcall was registered by the PH client.

Output Parameters

None.

Return Values

- `NPF_NO_ERROR`: The de-registration was successful.
- `NPF_PH_E_BAD_CALLBACK_HANDLE`: The API implementation does not recognize the callback handle. There is no effect to the registered callback functions.

6.9 Function to Get Number of Priorities Supported

Syntax

```
void NPF_PHNumPrioritiesGet(
     NPF_IN     NPF_callbackHandle_t   phCallbackHandle,
     NPF_IN     NPF_correlator_t       correlator,
     NPF_IN     NPF_errorReporting_t   phErrorReporting
);
```

Description of function

This function is used to retrieve the number of priority levels supported by the PH API implementation.

Input Parameters

- `phCallbackHandle:` Unique callback handle obtained during callback function registration.
- `Correlator:` The application's correlator for this call. It is used to distinguish between multiple invocations of the same API function call.
- `phErrorReporting:` This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.

Output Parameters

None.

Return Codes

- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle.

Asynchronous Response

The asynchronous response will contain the error code representing status of the call. If the error code in the callback indicates success, the number of supported priority levels will be passed to the callback function.

6.10 Function to Get Packet Handler Statistics

Syntax

```
void NPF_PHStatisticsGet(
     NPF_IN     NPF_callbackHandle_t   phCallbackHandle,
     NPF_IN     NPF_correlator_t       correlator,
     NPF_IN     NPF_errorReporting_t   phErrorReporting
);
```

Description of function

This function is used to retrieve the statistics maintained by the PH API. The number of statistics counters that are filled up by the PH API implementation depends on how many queues are supported in the implementation. If the number of queues supported is less that the number of priorities (8), then the indices that return valid counter values should be the left elements of the middle column of Table 1 in Section 4.2.6. This is to aid inter operability between PH API implementations and also to allow for the same relative treatment of priorities among different implementations.

Input Parameters

- `phCallbackHandle:` Unique callback handle obtained during callback function registration.
- `Correlator:` The application's correlator for this call. It is used to distinguish between multiple invocations of the same API function call.
- `phErrorReporting:` This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.

Output Parameters

None.

Return Codes

- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle.

Asynchronous Response

The asynchronous response will contain the error code representing status of the call. If the error code in the callback indicates success, a pointer to the PH statistics structure `NPF_phStatistics_t` will be passed to the callback function.

6.11 Optional Function to Create a Send Flow

Syntax

```
NPF_error_t NPF_PHSendFlowCreate(
      NPF_IN     NPF_callbackHandle_t  phCallbackHandle,
      NPF_IN     NPF_correlator_t      phCbCorrelator,
      NPF_IN     NPF_errorReporting_t  phErrorReporting,
      NPF_IN     NPF_phSendFlow_t      *phSendFlow
);
```

Description of function

This function is used by an application to create a send flow specification. This is an asynchronous function and will result in a callback subsequently. The API implementation might have to initialize some state on the forwarding plane, set up queues (if it implements queues for packets with different priorities), etc. A unique sending flow handle will be returned to the application in the callback if there were no errors. When the application subsequently sends a packet using the `NPF_PHPacketSend()` function, it uses this handle as a parameter to indicate which flow the packet belongs to. Implementation of this function is optional.

Input Parameters

- `phCallbackHandle:` Unique callback handle obtained during callback function registration.
- `phCbCorrelator:` This is used by the application to distinguish between multiple invocations of the same call. This value is not interpreted by the API implementation and will be returned to the application in the callback.
- `phErrorReporting:` This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.
- `phSendFlow:` Pointer to a Packet Handler Send Flow specification, as defined in `NPF_phSendFlow_t`.

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The send flow was accepted successfully.
- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle.
- `NPF_PH_E_UNKNOWN_PROTOCOL_TYPE:` The protocol indicated in the Send Flow structure NPF_phSendFlow_t is unsupported.
- `NPF_PH_E_INVALID_PRIORITY:` The priority indicated in `NPF_phSendFlow_t` is unsupported.
- `NPF_PH_E_BAD_FIB_HANDLE:` The FIB handle indicated in `NPF_phSendFlow_t` is not a valid FIB handle.
- `NPF_PH_E_BAD_NEXTHOP_IDENTIFIER:` The next hop identifier indicated in `NPF_phSendFlow_t` is invalid.
- `NPF_PH_E_BAD_LOGICAL_LINK_IDENTIFIER:` The interface handle indicated in `NPF_phSendFlow_t` is invalid.
- `NPF_PH_E_UNSUPPORTED_FLAGS:` The send flags indicated in `NPF_phSendFlow_t` are not valid.

Asynchronous Response

The asynchronous response will contain the error code representing status of the call. A send flow handle corresponding to the flow created will be passed to the callback function.

6.12 Optional Function to Delete a Send Flow

Syntax

```
NPF_error_t NPF_PHSendFlowDelete (
      NPF_IN     NPF_callbackHandle_t    phCallbackHandle,
      NPF_IN     NPF_correlator_t        phCbCorrelator,
      NPF_IN     NPF_errorReporting_t    phErrorReporting,
      NPF_IN     NPF_phSendFlowHandle_t  phSendFlowHandle
);
```

Description of function

This function is used by an application to delete a send flow specification. The API implementation might have to clean up some state in the forwarding plane, etc. This is an asynchronous function and will result in a callback subsequently. Implementation of this function is optional.

Input Parameters

- `phSendFlowHandle:` This is a unique send flow handle returned to the application during a call to `NPF_PHSendFlowCreate()`.
- `phCbcorrelator:` The application's correlator for this call. It is used to distinguish between multiple invocations of the same API function call.
- `phErrorReporting:` This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The send flow was deleted successfully.
- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle.
- `NPF_PH_E_INVALID_SEND_FLOW_HANDLE:` The send flow handle is invalid. This does not affect currently created send flows.

6.13 Optional Function to Send a Packet using Send Flow Handle

Syntax

```
NPF_error_t NPF_PHPacketSend(
      NPF_IN     NPF_callbackHandle_t   phCallbackHandle,
      NPF_IN     NPF_correlator_t       correlator,
      NPF_IN     NPF_errorReporting_t   phErrorReporting,
      NPF_IN     NPF_phBufDescr_t       *packet,
      NPF_IN     NPF_phSendFlowHandle_t phSendFlowHandle
);
```

Description of function

This function is used by an application to send a packet. The PH API implementation cannot guarantee that the packet actually will be transmitted eventually. This is due to the fact that an error could occur at any stage of sending – L2 address resolution, physical error, etc. Hence, a successful return from this function only implies that the packet was received successfully and the API implementation will attempt transmission to the forwarding plane. Implementation of this function is optional.

Input Parameters

- `phCallbackHandle`: Unique callback handle obtained during callback function registration.
- `correlator`: The application's correlator for this call. It is used to distinguish between multiple invocations of the same API function call.
- `phErrorReporting`: This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.
- `packet`: This is the packet to be transmitted.
- `phSendFlowHandle`: Handle that specifies a particular send flow. This handle is obtained in the asynchronous response that results from a call to `NPF_PHSendFlowCreate`().

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR`: The packet was accepted successfully for transmission.
- `NPF_PH_E_BAD_CALLBACK_HANDLE`: The API implementation does not recognize the callback handle.
- `NPF_PH_E_INVALID_SEND_FLOW_HANDLE`: The send flow handle is unrecognized.
- `NPF_PH_E_TX_RESOURCE_UNAVAILABLE`: The PH API implementation did not have enough resources (space in the queues for the priority, etc.) for transmitting the packet.
- `NPF_PH_E_INVALID_PRIORITY`: The priority indicated in the flow information associated with the `NPF_phSendFlowHandle_t` is unsupported.
- `NPF_PH_E_BAD_FIB_HANDLE`: The FIB handle indicated in the flow information associated with the `NPF_phSendFlowHandle_t` is not a valid FIB handle.
- `NPF_PH_E_BAD_NEXTHOP_IDENTIFIER`: The next hop identifier indicated in the flow information associated with the `NPF_phSendFlowHandle_t` is invalid.
- `NPF_PH_E_UNSUPPORTED_FLAGS`: The send flags indicated in the flow information associated with the `NPF_phSendFlowHandle_t` are not valid.

- `NPF_PH_E_PARAMETER_NOT_SUPPORTED`: The PH API implementation does not support one of the parameters being passed in. This could happen, for instance, if the Next Hop information being passed in is different from what is supported by the NPF IPv4 Unicast API implementation running on the system.

6.14 Optional Receive Flow Function (upcall)

Syntax

```
typedef void (*NPF_PHRecvFlowFunc_t) (
     NPF_IN     NPF_phRecvFlowHandle_t   phRecvFlowHandle,
     NPF_IN     NPF_phBufDescr_t         *packet,
     NPF_IN     NPF_phRecvMetadata_t     *recvMetadata
);
```

Description of function

This is the upcall function that will be invoked by the PH API implementation into the PH client when it receives a packet that matches the registered flow from the forwarding plane. The API implementation also passes up metadata information about the packet that is being delivered. It is to be noted that that receive flows might intersect between applications (or between flows registered by the same application); if so, the API implementation would have to de-multiplex the packet into all registered consumers, possibly by performing copies of the packet.

Input Parameters

- `phRecvFlowHandle:` The receive flow handle that was created earlier by the PH client by a call to `NPF_PHRecvFlowRegister ().`
- `Packet:` Pointer to the packet that matched the receive flow.
- `recvMetadata:` The receive metadata for the packet that matched the receive flow.

Output Parameters

None.

Return Codes

None.

6.15 Optional Function to Register a Receive Flow Specification

Syntax

```
NPF_error_t NPF_PHRecvPacketFlowRegister(
      NPF_IN     NPF_callbackHandle_t  phCallbackHandle,
      NPF_IN     NPF_correlator_t      correlator,
      NPF_IN     NPF_errorReporting_t  phErrorReporting,
      NPF_IN     NPF_PHRecvFlowFunc_t  phRcvFlowFunc,
      NPF_IN     NPF_phRecvFlow_t      *phRcvFlow
);
```

Description of function

This function is used by an application to register a receive flow specification. This will allow the PH API implementation to perform de-multiplexing of packets into PH clients. The application can register multiple receive functions, each one associated with a particular receive flow. The API implementation will assign a unique handle that will be used later for de-registration of the receive flow specification. It is to be noted that this is an asynchronous function and the callback function associated with it will return the receive flow handle.

Input Parameters

- `phCallbackHandle:` The callback handle returned to the PH client during PH registration through the NPF_PHRegister() function.
- `correlator:` The application's correlator for this call. It is used to distinguish between multiple invocations of the same API function call.
- `phErrorReporting:` This is used by the application to indicate if it wishes to receive a completion callback or not, or only upon errors.
- `phRecvFlowFunc:` Pointer to the upcall function that will be invoked when the PH API delivers packets into the application that belong to the flow being registered.
- `phRecvFlow:` The receive flow being registered for.

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The registration was successful.
- `NPF_PH_E_BAD_CALLBACK_FUNC:` The callback function passed in was NULL.
- `NPF_PH_E_CALLBACK_ALREADY_REGISTERED:` No new registration was made since the receive flow specification was already registered. Whether this is treated as an error or not is dependent on the application.
- `NPF_PH_E_BAD_CALLBACK_HANDLE:` The API implementation does not recognize the callback handle.

6.16 Optional Function to Deregister a Receive Flow Specification

Syntax

```
NPF_error_t NPF_PHRecvPacketFlowDeregister(
      NPF_IN  NPF_phRecvFlowHandle_t phRecvFlowHandle
);
```

Description of function

This function is used by an application to deregister a receive flow specification. After this function returns successfully, the application cannot receive any packets that matched the flow specification. It is to be noted that this is a synchronous function and has no callback function associated with it.

In Parameters

- `phRecvFlowHandle:` The unique identifier representing the receive flow. This identifier was obtained during a previous call to NPF_PHRecvFlowRegister().

Output Parameters

None.

Return Codes

- `NPF_NO_ERROR:` The de-registration was successful.
- `NPF_PH_E_BAD_HANDLE:` The API implementation does not recognize the handle. There is no effect to the previous registration.

7 API Summary

The following table illustrates which parameters (metadata and packet) are present in the send and receive directions. An entry marked "X" means that the parameter is required for the specified direction. For example, when receiving packets, metadata that accompanies the packet includes priority, protocol type, interface/logical link it arrived on, reason code and reason sub-code. Also, no flags are giving into the application in the receive direction.

Metadata	Transmit	Receive
Packet	X	X
Buffer Descriptor	X	X
Priority	X	X
Flags	X	
Reason Code		X
Reason Sub code		X

The following table summarizes the NPF Packet Handler API.

API Name	API Required
NPF_PHRegister	YES
NPF_PHDeregister	YES
NPF_PHEventRegister	YES
NPF_PHEventDeregister	YES
NPF_PHPacketSendTo	YES
NPF_PHRecvPacketRegister	YES
NPF_PHRecvPacketDeregister	YES
NPF_PHNumPrioritiesGet	YES
NPF_PHStatisticsGet	YES
NPF_PHSendFlowCreate	NO
NPF_PHSendFlowDelete	NO
NPF_PHPacketSend	NO
NPF_PHRecvFlowRegister	NO
NPF_PHRecvFlowDeregister	NO

Table 2: Summary of Packet Handler API

8 References

[1] NP Forum - Software API Framework Lexicon Implementation Agreement Revision 1.0

[2] NP Forum – Software API Conventions Implementation Agreement Revision 2.0

[3] NP Forum – Software API Framework Implementation Agreement Revision 1.0

[4] NP Forum – Interface Management API Implementation Agreement Revision 1.0

[5] NP Forum - IPv4 Unicast forwarding API (latest draft)

[6] NP Forum – IPv6 forwarding API (latest draft)

[7] IANA (http://www.iana.org/assignments/ethernet) – EtherType numbers

[8] IANA (http://www.iana.org/assignments/ppp-numbers) – PPP numbers

[9] IANA (http://www.iana.org/assignments/protocol-numbers) – IP Protocol numbers

[10] IANA (http://www.iana.org/assignments/ieee-802-numbers) – IEEE 802 LSAP

Appendix A Header File - NPF_PH_API.H

```
/*
 * This header file defines typedefs, constants and
 * functions of the NPF Packet Handler API
 *
 * This assumes that the definitions common to all NPF APIs
 * are available in a separate manner (a different header
 * file, etc.)
 */

#ifndef __NPF_PH_API_H_
#define __NPF_PH_API_H_

#ifdef __cplusplus
extern "C" {
#endif

/**********************************
* NPF Buffer descriptor structure.
*

1. Buffer descriptor containing pointer and length.
(a) Pointer is either to the first byte of the packet, or to some structure
chosen or defined by the implementer.
(b) Length is the total length of the packet, in bytes.
**********************************/

typedef struct {
      NPF_uint8_t *packetStart;
      /* pointer to first byte of packet OR some structure
         containing the first byte of the packet */
      NPF_uint32_t packetLen;   /* payload length */
} NPF_phBufDescr_t;

typedef NPF_uint32_t NPF_phOffset_t;

/**********************************
* Protocol Type Indicator
This structure identifies a protocol by its protocol number.   Protocol
numbers are assigned by several different     numbering authorities, so there
is a protocol family      identifier to say what number series the protocol
number          belongs to.   The IEEE and IETF number assignments can be
found on the IANA web site,http://www.iana.org/numbers.html.
    **********************************/

typedef enum {
      NPF_PROTO_FAM_LLC = 1,/* IEEE 802 LSAP assignments */
      NPF_PROTO_FAM_ET = 2,/* IEEE EtherType assignments */
      NPF_PROTO_FAM_PPP = 3,/* IETF PPP protocol numbers */
      NPF_PROTO_FAM_IP = 4, /* IETF IP protocol numbers */
```

```
      NPF_PROTO_FAM_NPF = 5,  /* NP Forum-defined numbers
                          */
      NPF_PROTO_FAM_UNK = 6,  /* Unknown protocol */
      NPF_PROTO_FAM_ANY = 7   /* Wildcard value */
} NPF_protocolFamily_t;

typedef struct {
      NPF_protocolFamily_t    family;
      NPF_uint16_t            protocol;
} NPF_protocol_t;

/**********************************
* Priority
Priority is a metadata element passed with both receive and transmit packets.
It can be used by an API implementation     that has packet queues within it:
if its queues become      congested, it can relieve congestion by discarding
packets        in order of priority, lowest priority first.   There is no
other use intended for this priority value.
**********************************/
typedef NPF_uint8_t NPF_phPriority_t;

#define NPF_PH_PRIORITY_LOWEST 0
#define NPF_PH_PRIORITY_HIGHEST 7

/**********************************
* Send Flags
The Send Flags metadata element contains boolean flags that
can condition the operation of the FE for sending specific
kinds of packets.  Currently the only flag defined is:

Locally Sourced -- means the packet originates from the local host system,
and should be forwarded as such.  This can be used by the PH implementation
and the FE together to make sure the TTL is not decremented by the FE, that
input   firewall   filters   are   bypassed,   etc.,   as   appropriate   to   the
implementation.
**********************************/

typedef NPF_uint32_t NPF_phSendFlags_t;

#define NPF_PH_SEND_FLAGS_LOCAL_SOURCE 0x00000001

/**********************************
*     Reason Code and Subcode

These metadata can be used where the FE is capable of indicating a reason
that it chose to send a packet to the Packet Handler.  The subcode metadata
may contain more specific information about the reason for diverting the
packet.
**********************************/
typedef enum
{
      NPF_PH_RECV_REASON_CODE_NO_DEST_ADDR = 1,
            /*destination address was not found */
```

```
      NPF_PH_RECV_REASON_CODE_LOCALLY_ADDRESSED = 2,
            /* locally destined packet */
      NPF_PH_RECV_REASON_CODE_UNSUPPORTED_OPTION = 3,
            /* Unsupported option */
      NPF_PH_RECV_REASON_CODE_UNSUPPORTED_PROTOCOL = 4,
            /* Unknown protocol */
      NPF_PH_RECV_REASON_CODE_FILTER_MATCH = 5
            /* Matched a filter */
    } NPF_phRecvReasonCode_t;

typedef NPF_uint32_t NPF_phRecvReasonSubcode_t;

#define NPF_PH_RCV_REASON_SUBCODE_ANY 0

/*********************************
Logical Link Identifier Structure

A logical link is like an interface, but more specific.  It
identifies an input or output link as a LAN port, a point-
to- point interface like POS, or, for subdivided interfaces
like ATM, it indicates a specific VPI/VCI.  It can be
extended to indicate a specific MPLS path as well.
*********************************/
typedef struct
{
      NPF_IfHandle_t  interfaceHandle;
      NPF_IfType_t      intfType;
      union
      {
          NPF_uint32_t  unused;
          NPF_VccAddr_t vpiVci;
      } u;
} NPF_LogicalLinkID_t;

#define NPF_INTERFACE_HANDLE_ANY    0

/* FIB handles from NPF IPv4 API and NPF IPv6 API*/
typedef NPF_uint32_t NPF_IPv4UC_FwdTableHandle_t;
typedef NPF_uint32_t NPF_IPv6UC_FwdTableHandle_t;

/*********************************
Sending Specification

The Sending Specification is metadata that accompanies
output packets. In various ways, it can convey information
to the FE about how a forwarding decision should be made
for this packet.  The possibilities are (mutually
exclusive):
  - No specification given
  - Send the packet out a specific logical link (i.e.,
interface,
    ATM connection, MPLS path, etc.)
```

```
  - Use a particular forwarding table in a case where the
FE
    does a destination address lookup to forward the packet
  - Use a particular next hop (interface and next hop IP
address).
It can be extended to include other protocols that may need
to be supported.
**********************************/
typedef enum
{
      NPF_PH_SEND_SPEC_NONE = 1,
      NPF_PH_SEND_SPEC_TYPE_OUTPUT_LINK_IDENTIFIER = 2,
      NPF_PH_SEND_SPEC_TYPE_IPv4FIB_IDENTIFIER = 3,
      NPF_PH_SEND_SPEC_TYPE_IPv4NEXTHOP_IDENTIFIER = 4,
      NPF_PH_SEND_SPEC_TYPE_IPv4NEXTHOP_INFO = 5,
      NPF_PH_SEND_SPEC_TYPE_IPv6FIB_IDENTIFIER = 6,
      NPF_PH_SEND_SPEC_TYPE_IPv6NEXTHOP_IDENTIFIER = 7,
      NPF_PH_SEND_SPEC_TYPE_IPv6NEXTHOP_INFO = 8
} NPF_phSendSpecType_t;

typedef struct
{
      NPF_IPv4Address_t nextHopAddr;
      NPF_IfHandle_t      interfaceHandle;
} NPF_IPv4NextHopInfo_t;

typedef struct
{
      NPF_IPv6Address_t nextHopAddr;
      NPF_IfHandle_t          interfaceHandle;
} NPF_IPv6NextHopInfo_t;

typedef struct
{
      NPF_phSendSpecType_t      SendSpecType;
      union
      {
          NPF_uint32_t                unused;
          NPF_LogicalLinkID_t         logicalLink;
          NPF_IPv4UC_FwdTableHandle_t v4fibHandle;
          NPF_IPv6UC_FwdTableHandle_t v6fibHandle;
          NPF_uint32_t                nextHopIdentifier;
          NPF_IPv4NextHopInfo_t       v4nextHop;
          NPF_IPv6NextHopInfo_t       v6nextHop;
      } u;
} NPF_phSendSpec_t;

/*************************************************
Send Metadata

This structure contains all the metadata needed for sending
a packet.
```

```
It contains:
- protocol - this is the header protocol type
- Priority - Priority of the packet (if supported by the implementation)
- Sending Specification - Routing hints for sending the packet out.
- Send Flags - Locally sourced, etc.
************************************************/
typedef struct
{
      NPF_protocol_t          protocolType;
      NPF_phPriority_t        priority;
      NPF_phSendSpec_t        sendSpec;
      NPF_phSendFlags_t       sendFlags;
} NPF_phSendMetadata_t;

/**********************************
Receive Metadata

Receive Metadata is the structure passed up to the client with a received
packet.  It contains the following:

Priority – priority that the packet was treated with in the PH API
implementation
Reason code and sub code – indicates why the packet is being delivered to the
PH client
Logical Link Identifier – This identifies the logical link (interface, ATM
connection, MPLS path, etc.) on which the packet arrived. If the client needs
to know the protocol identity at the first byte of the packet, it must be
able to infer this value from the logical link. If the NPE  removes one or
more headers, this  logical  link  identifier  must permit the client to
identify the  protocol of the header appearing first in the buffer (the first
header  following the ones that were removed)
Offset  information  -  contains  a  protocol  ID  and  offset  value.  These
parameters are only used when the NPE has performed some packet processing
already and wishes to share with the API client some additional information.
      (a) Offset is a number ranging from zero to length-1.  At the offset
(bytes) from the beginning the packet is the first byte of a protocol header
identified by the Protocol ID.
      (b) The protocol ID of the header indicated at the offset. Note that if
offset is zero, this value is also zero, as the protocol is inferred from the
logical link identifier.
**********************************/

typedef struct
{
      NPF_uint32_t                offset;
      NPF_protocol_t              protocolID;
      NPF_LogicalLinkID_t         logicalLink;
      NPF_phRecvReasonCode_t      reasonCode;
      NPF_phRecvReasonSubcode_t   reasonSubCode;
      NPF_phPriority_t            priority;
} NPF_phRecvMetadata_t;
```

```
/*************************************************
Send Flow Specification

This structure contains all the metadata needed for sending a packet. It is
used when creating a send flow.

It has the same contents as the Send Metadata
*************************************************/
typedef NPF_phSendMetadata_t NPF_phSendFlow_t;


/*********************************
Send Flow Handle

The PH API client supplies sending metadata when it sends
packets via the Packet Handler.  If it calls the
NPF_PHPacketSendTo() function, it supplies metadata each time it sends a
packet.  As an optimization, the API defines the concept of a Send Flow --
shorthand for metadata that is
the same for every packet in the flow.  The client supplies
the sending metadata when it creates a flow.  The API
implementation stores the metadata and returns a flow
handle, which the client can use when sending packets using
the NPF_PHPacketSend() function.
*********************************/
typedef NPF_uint32_t NPF_phSendFlowHandle_t;

/*********************************
Receive Flow Specification

Applications register interest in packet flows by
specifying receive flows. The API implementation returns a
Recv Flow handle that will be used subsequently when the
API implementation hand up packets to the application.  The
contents are as follows:

Protocol Type -- matches the protocol type of the first
header in the packet, as passed up from the FE.  A value of
NPF_PROTO_FAM_ANY in the protocol family variable matches
ANY protocol.

Logical Link -- Specifies the logical link on which the
packet was received.  If the Interface Handle part of this
is null, it matches any logical link.

Reason Code -- Specifies the reason code to match.  A value
of NPF_PH_RCV_REASON_CODE_ANY matches any reason code.

Reason Subcode -- specifies the reason subcode to match, if
the given reason code uses a subcode.  NPF_PH_RCV_REASON_SUBCODE_ANY is a
wildcard
value.
```

```
*********************************************************/

typedef struct
{
      NPF_protocol_t                protocolType;
      NPF_LogicalLinkID_t           logicalLink;
      NPF_phRecvReasonCode_t        reasonCode;
      NPF_phRecvReasonSubcode_t     reasonSubCode;
} NPF_phRecvFlow_t;

/*********************************
Receive Flow Handle

When a client registers a receive flow, the API implementation returns a
handle for it. This is used later or deregistering the receive flow.
*********************************/

typedef NPF_uint32_t NPF_phRecvFlowHandle_t;

/***********************************************
Packet Handler Statistics
Each priority that is supported has 8 counters
This accounts for 8*6 = 48 counters.
In addition, the 49th counter is for packets recvd that
were dropped if there were no flows registered for it
***********************************************/

typedef struct
{
NPF_uint32_t npfPHPktsTx[8];
NPF_uint32_t npfPHBytesTx[8];
NPF_uint32_t npfPHTxPktsDropped[8];
NPF_uint32_t npfPHPktsRx[8];
NPF_uint32_t npfPHBytesRx[8];
NPF_uint32_t npfPHRxPktsDropped[8];
NPF_uint32_t npfPHRcvPktsDroppedNoFlow;
} NPF_phStatistics_t;

typedef enum
{
      NPF_PH_CB_TYPE_SEND_PACKET = 1,
      NPF_PH_CB_TYPE_CREATE_SEND_FLOW = 2,
      NPF_PH_CB_TYPE_SEND_PACKET_TO = 3,
      NPF_PH_CB_TYPE_DEL_SEND_FLOW = 4,
      NPF_PH_CB_TYPE_REG_RCV_FLOW = 5,
      NPF_PH_CB_TYPE_GET_NUM_PRIORITIES = 6,
      NPF_PH_CB_TYPE_GET_STATISTICS = 7
} NPF_phCallbackType_t;

typedef NPF_uint32_t NPF_phErrorType_t;

#define NPF_PH_BASE_ERR (NPF_INTERFACES_MAX_ERR + 1)
```

```
#define NPF_PH_MAX_ERR (NPF_PH_BASE_ERR + 99)

#define NPF_PH_E_BAD_CALLBACK_FUNCTION    (NPF_PH_BASE_ERR+1)
#define NPF_PH_E_CALLBACK_ALREADY_REGISTERED (NPF_PH_BASE_ERR+2)
#define NPF_PH_E_BAD_CALLBACK_HANDLE (NPF_PH_BASE_ERR+3)
#define NPF_PH_E_UNKNOWN_PROTOCOL (NPF_PH_BASE_ERR+4)
#define NPF_PH_E_UNSUPPORTED_OPTION (NPF_PH_BASE_ERR+5)
#define NPF_PH_E_BAD_FIB_ID (NPF_PH_BASE_ERR+6)
#define NPF_PH_E_BAD_NEXTHOP_IDENTIFIER (NPF_PH_BASE_ERR+7)
#define NPF_PH_E_BAD_LOGICAL_LINK_IDENTIFIER (NPF_PH_BASE_ERR+8)
#define NPF_PH_E_UNSUPPORTED_FLAGS (NPF_PH_BASE_ERR+9)
#define NPF_PH_E_INVALID_SEND_FLOW_HANDLE (NPF_PH_BASE_ERR+10)
#define NPF_PH_E_INVALID_PRIORITY (NPF_PH_BASE_ERR+11)
#define NPF_PH_E_BAD_BUFFER (NPF_PH_BASE_ERR+12)
#define NPF_PH_E_TX_RESOURCE_UNAVAILABLE (NPF_PH_BASE_ERR+13)
#define NPF_PH_E_PARAMETER_NOT_SUPPORTED (NPF_PH_BASE_ERR+14)

/* Error codes from sending packet out of the device
   These errors can be reported only if the device informs
   the PH API implementation of these errors during transmission
*/
#define NPF_PH_E_SEND_FAIL (NPF_PH_BASE_ERR+15)
#define NPF_PH_E_ARP_FAIL (NPF_PH_BASE_ERR+16)
#define NPF_PH_E_INVALID_PORT (NPF_PH_BASE_ERR+17)
#define NPF_PH_E_PHYSICAL_ERROR (NPF_PH_BASE_ERR+18)
#define NPF_PH_E_INTERNAL_ERROR (NPF_PH_BASE_ERR+19)

/***********************************************
Callback Structure
This is delivered into the application for every API call
it makes. The Callback type can be used by the application
to demultiplex the response. The union contains the
corresponding response data
***********************************************/
typedef struct
{
      NPF_phCallbackType_t    type;
      NPF_phErrorType_t       error;
      union
      {
          NPF_uint32_t              unused;
          NPF_phBufDescr_t         *bufDescr;
          NPF_uint32_t              phNumPriorities;
          NPF_phStatistics_t       *phStatistics;
          NPF_phSendFlowHandle_t    phSendFlowHandle;
          NPF_phRecvFlowHandle_t    phRecvFlowHandle;
      } u;
} NPF_phCallbackData_t;

/*
      Event types
*/
```

```
typedef enum {
      NPF_PH_TX_RESOURCE_AVAILABLE = 0,
} NPF_phEvent_t;

/*
      Data structures for event callback
*/

/* This structure reports the minimum priority level
available for a PH API client for transmission. An array of
this structure can be reported to the client.
*/
typedef struct NPF_PHEventData {
      NPF_phEvent_t     phEventType; /* PH Event */
      union {
            NPF_uint16_t      minPriority; /* Minimum priority
                        level available for transmission */
      } u;
} NPF_phEventData_t;

typedef struct _PHEventArray_t {
      NPF_uint16_t      n_data; /* Number of events in array */
      NPF_phEventData_t *eventData; /* Array of events */
} NPF_phEventArray_t;

/*****************************************************
 *          PACKET HANDLER API FUNCTION CALLS            *
 *****************************************************/
typedef void (*NPF_phCallbackFunc_t)(
      NPF_userContext_t       userContext,
      NPF_correlator_t        phCorrelator,
      NPF_phCallbackData_t    *phCallbackdata
);

typedef void (*NPF_phEventCallFunc_t) (
      NPF_userContext_t       userContext,
      NPF_phEventArray_t      phEventArray);

NPF_error_t NPF_PHRegister(
      NPF_userContext_t       userContext,
      NPF_phCallbackFunc_t    phCallbackFunc,
      NPF_callbackHandle_t    *phCallbackHandle
);

NPF_error_t NPF_PHDeregister(
      NPF_callbackHandle_t phCallbackHandle
      );

NPF_error_t NPF_PHEventRegister(
      NPF_userContext_t       userContext,
      NPF_phEventCallFunc_t   phEventHandlerFunc,
      NPF_callHandle_t        *phEventHandle
```

```
);

NPF_error_t NPF_PHEventDeregister(
      NPF_callHandle_t        phEventHandle
      );

NPF_error_t NPF_PHPacketSendTo(
      NPF_callbackHandle_t    phCallbackHandle,
      NPF_correlator_t        correlator,
      NPF_errorReporting_t    phErrorReporting,
      NPF_phBufDescr_t        *packet,
      NPF_phSendMetadata_t    *phSendMetadata
);

typedef void (*NPF_PHRecvPacketFunc_t) (
      NPF_callbackHandle_t    phRcvCallbackHandle,
      NPF_phBufDescr_t        *packet,
      NPF_phRecvMetadata_t    *phRecvMetadata
);

NPF_error_t NPF_PHRecvPacketRegister(
      NPF_userContext_t       userContext,
      NPF_PHRecvPacketFunc_t  phRcvPktFunc,
      NPF_callbackHandle_t    *phRcvCallbackHandle
);

NPF_error_t NPF_PHRecvPacketDeregister(
      NPF_callbackHandle_t    phRcvCallbackHandle
);

void NPF_PHStatisticsGet(
      NPF_callbackHandle_t    phCallbackHandle,
      NPF_correlator_t        correlator,
      NPF_errorReporting_t    phErrorReporting
);

NPF_error_t NPF_PHSendFlowCreate(
      NPF_callbackHandle_t    phCallbackHandle,
      NPF_correlator_t        phCbCorrelator,
      NPF_errorReporting_t    phErrorReporting,
      NPF_phSendFlow_t        *phSendFlow
);

NPF_error_t NPF_PHSendFlowDelete (
      NPF_callbackHandle_t    phCallbackHandle,
      NPF_correlator_t        phCbCorrelator,
      NPF_errorReporting_t    phErrorReporting,
      NPF_phSendFlowHandle_t  phSendFlowHandle
);

NPF_error_t NPF_PHPacketSend(
      NPF_callbackHandle_t    phCallbackHandle,
```

```
      NPF_correlator_t        correlator,
      NPF_errorReporting_t    phErrorReporting,
      NPF_phBufDescr_t        *packet,
      NPF_phSendFlowHandle_t  phSendFlowHandle
);

typedef void (*NPF_PHRecvFlowFunc_t) (
      NPF_phRecvFlowHandle_t  phRecvFlowHandle,
      NPF_phBufDescr_t        *packet,
      NPF_phRecvMetadata_t    *phRecvMetadata
);

NPF_error_t NPF_PHRecvPacketFlowRegister(
      NPF_callbackHandle_t    cbHandle,
      NPF_correlator_t        correlator,
      NPF_errorReporting_t    phErrorReporting,
      NPF_PHRecvFlowFunc_t    phRcvFlowFunc,
      NPF_phRecvFlow_t        *phRcvFlow
);

NPF_error_t NPF_PHRecvPacketFlowDeregister(
      NPF_phRecvFlowHandle_t  phRecvFlowHandle
);

#ifdef __cplusplus
}
#endif

#endif /* __NPF_PH_API_H_ */
```

Appendix B List of companies belonging to NPF during approval process

Agere Systems	IBM	Samsung Electronics
Alcatel	IDT	Sandburst Corporation
Altera	Intel	Silicon & Software Systems
AMCC	IP Infusion	Silicon Access
Analog Devices	Kawasaki LSI	Sony Electronics
Avici Systems	LSI Logic	STMicroelectronics
Azanda Network Devices	Modelware	Sun Microsystems
Cypress Semiconductor	Mosaid	Teja Technologies
Ericsson	Motorola	TranSwitch
Erlang Technologies	NEC	U4EA Group
EZ Chip	NetLogic	Xelerated
Flextronics	Nokia	Xilinx
Fujitsu Ltd.	Paion Co., Ltd.	Zettacom
FutureSoft	PMC Sierra	ZTE
HCL Technologies	RadiSys	Hi/fn