IPv6 Unicast Forwarding Service API Implementation Agreement

Revision 1.0

Editor(s):

Kristian Gregersen, kristian.gregersen@ericsson.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	09/25/2003	Created Rev 1.0 of the implementation agreement by taking the IPv6 Unicast Forwarding Service API (npf2002.616.05) and making minor editorial corrections.

2 Introduction

One prevalent use of network processors is the implementation of devices that perform packet forwarding based on IPv6 unicast destination addresses. There are at least two databases needed for IP forwarding, one being the Routing Information Base (RIB), which resides on the control plane, and the other being the Forwarding Information Base (FIB), which network processors may access on the forwarding plane.

Ingress packets have their destination IP address extracted and used as a lookup key in the FIB. Assuming a match is found, this forwarding information typically provides a next hop IP address and an egress interface, which can be used to reach this next hop. The next hop is usually the IP address of the router that provides a path to the final destination of the packet. The forwarding information located in the FIB may entail not only next hop information, but also other characteristics, such as QoS based on DiffServ, or encapsulation schemes, such as MPLS tunneling.

The RIB may be defined by static configuration or via dynamic routing protocols, such as OSPF and BGP. Often, such application layer software will interface with an intelligent Route Table Manager (RTM) component, whose job is to manage the RIB and maintain the FIB used by the forwarding plane IP packet handling. Usually, the RIB contains all the routes known to all routing protocols, and the FIB is the “active” subset of those routes – the ones chosen as best for forwarding. The RTM can use NPF defined IPv6 Unicast Forwarding Service API function calls to manage an IPv6 FIB.

Another component of IPv6 packet forwarding is the description of a method to resolve a next hop destination IPv6 address into the associated media address. For IPv6 the method used for this purpose is encompassed in the Neighbor Discovery protocol (ND, defined in RFC 2461). IPv6 ND uses IP multicast for transporting its IPv6 ICMP messages (ICMPv6, defined in RFC 2463). This is a major difference from IPv4, for which the Address Resolution method to resolve a nexthop destination IPv4 address into the associated media address is defined as a protocol that runs directly as a Layer 2 client, with peculiarities dependent on the L2 protocol.

In general, nodes (hosts and routers) use Neighbor Discovery to determine the link-layer addresses for neighbors known to reside on attached links and to quickly purge cached values that become invalid. Nodes use the protocol to actively keep track of which neighbors are reachable and which are not, and to detect changed link-layer addresses. Hosts use Neighbor Discovery to find neighboring routers that are willing to forward packets on their behalf. When a router or the path to a router fails, a host actively searches for functioning alternates.

Neighbor Discovery can be handled either as part of the control plane or realized below the SAPI. This contribution includes primitives to control an address resolution table. The address resolution table corresponds to the neighbor cache concept of RFC 2461.

A Neighbor Discovery handler will use the Packet Handler API for packet exchange.

A basic example system might have the following characteristics:

- An RTM managing a RIB in the control plane will use NPF IPv6 Unicast Forwarding Service API calls to maintain a FIB for use by a network processor.
- An NP in the forwarding plane has knowledge and control of one or more layer 3 interfaces.
- The NP has knowledge of, and access to, a FIB.
- Each FIB is associated with one or more layer 3 interfaces.
- The FIB has been created at some point and is referenced by a unique handle.
- One or more NP’s may be present in the system.

For example, the initial ingress forwarding steps with this model might be:

- A packet arrives on an interface.
- The FIB is selected based on the incoming layer 3 interface.
- If the packet’s destination address is a global IPv6 address, the longest prefix match lookup of the packet's destination IPv6 address is done using this particular FIB.

The following two figures may prove useful in understanding the specification of the IPv6 Unicast Forwarding Service API.

In Figure 1, the RTM oversees the management of routes provided by routing protocols and maintains a single RIB. Using the IPv6 Unicast Forwarding Service API, the RTM defines and populates the FIB, knowing only about one FIB which is identified by a unique FIB handle. A particular FIB may be replicated in different NP devices. Such replication may be done by the Services API implementation or by some system-aware middleware below it.

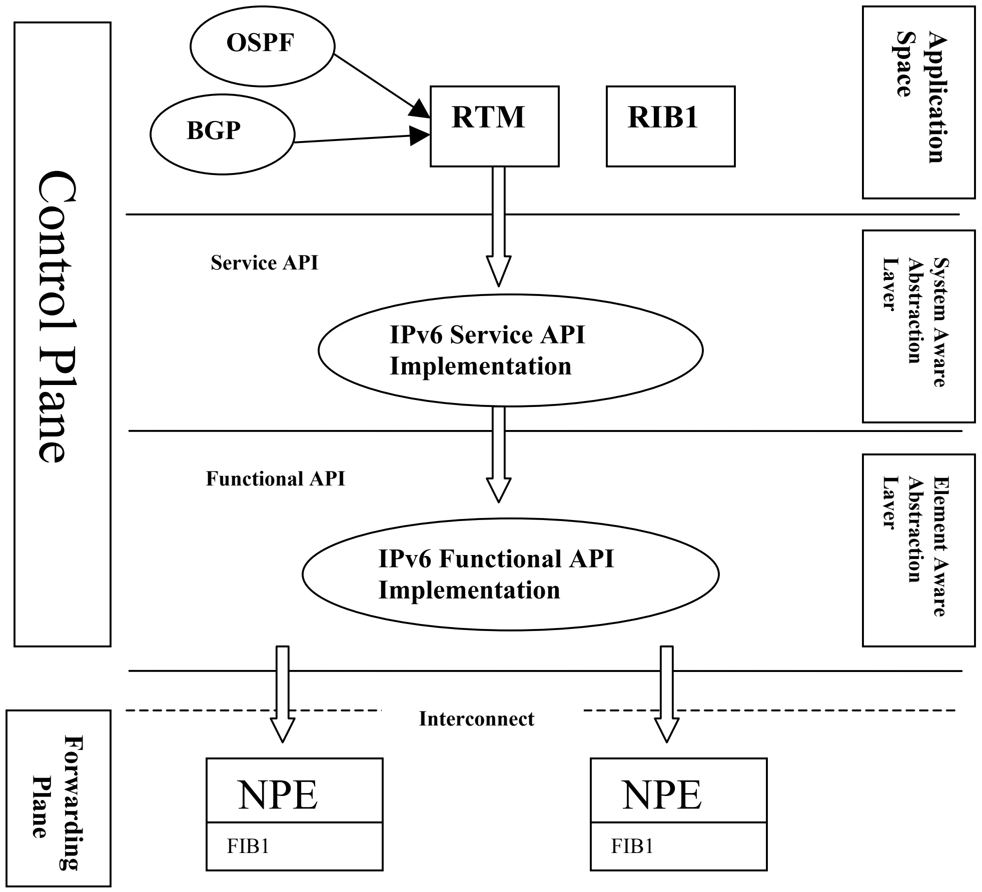
Control Plane
OSPF
BGP
RTM
RIB1
Application Space
Service API
IPv6 Service API Implementation
System Aware Abstraction Layer
Functional API
IPv6 Functional API Implementation
Element Aware Abstraction Layer
Interconnect
Forwarding Plane
NPE
FIB1
NPE
FIB1

Figure 1 - Example Single FIB System

Yet another design is worth consideration. This implementation might represent a system that has created multiple virtual routers in order to realize a Virtual Private Network. In this scheme, isolation is provided between routing domains by maintaining independent RIBs, and as a consequence, unique instances of their associated FIBs. In this situation, the control plane has knowledge of two unique FIBs and will be dealing with two unique FIB handles.

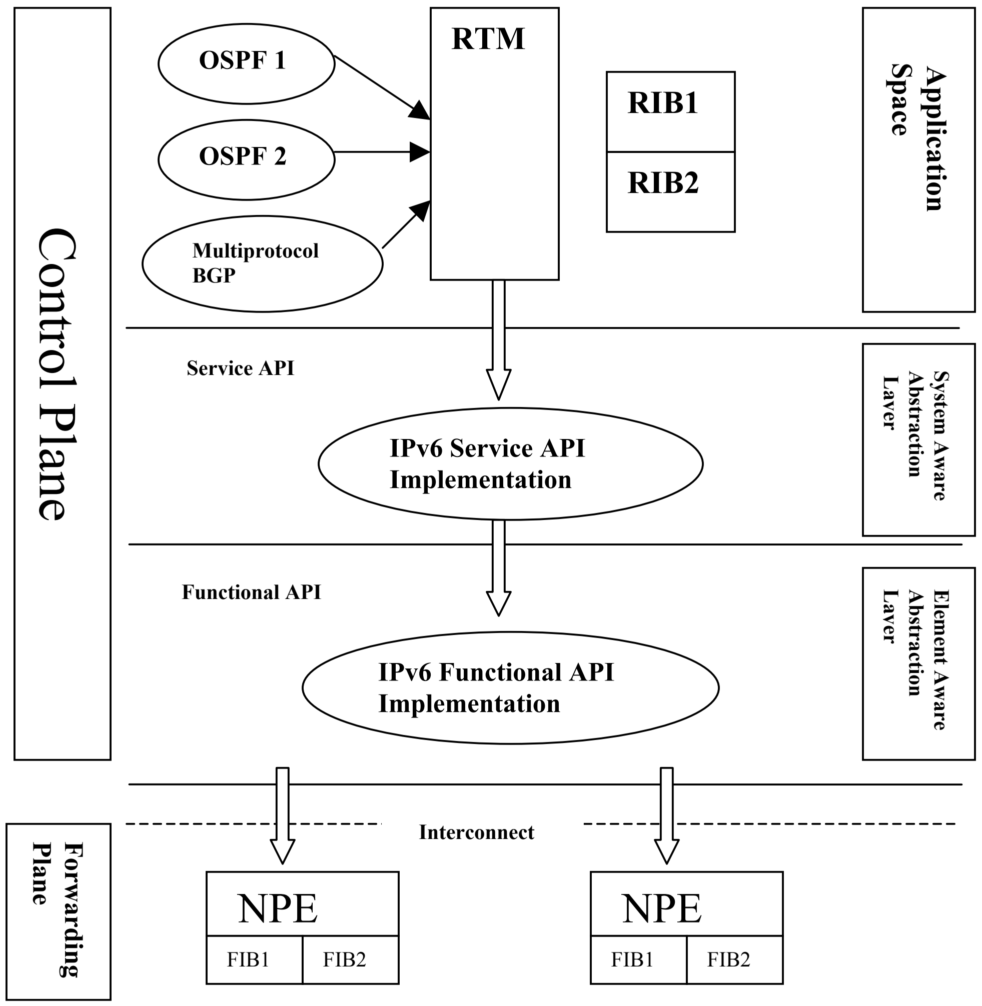

Figure 2 - Example Multiple FIB System

The introduction, so far, has presented high level concepts related to IPv6 Unicast Forwarding and the placement of various components. This document acknowledges the wide range and variety of target environments for control plane applications and NPs. In Section 3, *API Usage Model*, in depth information is provided regarding the representation of a FIB and the effect this has on the design of the IPv6 Unicast Forwarding Service API.

The remainder of this document is organized as follows:

- Section 3 describes forwarding information base models and usage of the corresponding API function calls. The unified and discrete models are covered as well as the optional address resolution.
- Section 4 describes the data structures, callbacks, return values, and events used in the IPv6 Unicast Forwarding Service API.
- Section 5 describes the function calls used in the IPv6 Unicast Forwarding Service API.
- Section 6 summarizes the function call names by category and also provides a list of events.
- Section 7 provides references to other relevant NPF documentation.
- Appendix A provides an informative appendix containing an IPv6 Unicast Forwarding header file.
- Appendix B provides a list of NPF members at the time of approval.

2.1 Assumptions and External Requirements

- For a better understanding of this specification, it is assumed that the reader has an understanding of the concepts and guidelines presented in the following NPF Software Implementation Agreements:
    - Software API Conventions (Revision 2, September 2003).
    - API Software Framework (Revision 1, August 2002).
    - Interface Management API (Revision 1, August 2002).
- While the term “table” is used throughout this document, this is a convenience to describe the model. There is no requirement that tables be implemented below the API, either on the control plane or the forwarding plane.
- The following concepts are contained in the NPF Software Implementation Agreement – Interface Management API (Revision 1, August 2002):
    - The description of how logical layer 3 interfaces are associated with a particular FIB.
- All API calls are considered asynchronous in nature, unless otherwise specified. The definitions of synchronous and asynchronous behavior are specified by the NPF Software Implementation Agreement – Software API Conventions (Revision 2, September 2003).
- As specified in the NPF Software Implementation Agreement – Software API Conventions (Revision 2, September 2003), Section 6.4, memories that are used to hold values that are passed as parameters are “owned” by the side that allocated them. An owner of a memory is responsible for de-allocating this memory when it is no longer used.

2.2 Scope

- This specification describes data structures for IPv6 unicast forwarding and address resolution. The data types and structures generally used by all API specifications are defined by the NPF Software Implementation Agreement - Software API Conventions (Revision 2, September 2003) document; however, IPv6 specific structures must be defined in an update of this document.

- This specification describes Service API definitions for IPv6 unicast forwarding and address resolution. The API function details will include input/output parameters, return code specifications and detailed usage notes specific to each invocation.
- This specification provides details regarding the handling of asynchronous events and expected responses from API function invocations, including specifications for completion callback and event handler routine registration and deregistration.

2.3 Dependencies

This document depends on the NPF Software Implementation Agreement - Software API Conventions (Revision 2, September 2003) document for basic type definitions.

The document also depends on an update of the NPF Software Implementation Agreement – Interface Management API (Revision 1, August 2002) document for the definition of NPF_IfHandle_t encompassing definition of IPv6 interfaces and other functions to manage IPv6 interfaces.

Furthermore, this document depends on the NPF Software Implementation Agreement – IPv4 Unicast Forwarding Service API for the definition of NPF_MediaAddress_t and NPF_MediaType_t.

3 API Usage Model

Depending upon the networking environment, control plane application design and forwarding plane NP architecture, a Forwarding Information Base (FIB) may be modeled in several ways. This document considers two modes for organizing and manipulating the IPv6 unicast forwarding information at the Service API level. Since it is customary to describe a Forwarding Information Base (FIB) in terms of a table data structure, this abstraction is used throughout the rest of this specification.

The first mode, called the *unified table mode*, uses a single table for structuring and managing IPv6 unicast forwarding information. The second mode, called the *discrete table mode*, uses separate prefix and next hop tables for structuring and managing IPv6 unicast forwarding information. Additionally, both modes represent address resolution information using a separate address resolution table.

The modes do not imply that the underlying NP forwarding elements support one or the other of these modes. Since this is an NPF Services API, the application has no knowledge of the individual NP forwarding elements, so the modes only specify the application layer interface to the IPv6 Unicast Forwarding services provided by the system. The models each represent a shared view between an application and the Service API implementation.

For example, an NP forwarding element may implement discrete mode prefix and next hop tables, whereas the control plane routing application may organize its FIB information in a unified model. In such a case, it is appropriate for the application to use the unified mode API function calls. The IPv6 Unicast Forwarding Services API implementation or some other software below it on the CP or NP forwarding element, would then be responsible for mapping the unified parameters to a suitable format for the discrete mode organization of the NP.

The two modes and their data entities are representative of a large number of system implementations. However, based on a desire for maximum interoperability and a perceived current market prevalence of routing applications designed using a unified mode, it is so stated:

Compliant implementations of the IPv6 Unicast Forwarding Service API specification MUST implement the *unified table mode,* but MAY also implement the *discrete table mode*, according to segment needs and product capabilities.

With the above requirements statement, it is acknowledged that certain combinations of application and NP architecture choices may place an extra burden upon either the application or the IPv6 Service API implementation. Therefore, the decision to offer an optional discrete mode is prompted primarily to alleviate two concerns:

- First, excessive amounts of storage may be required to maintain state information if the Service API mode does not match the underlying forwarding element representation. This is particularly important because many network processors have imbedded control points with limited storage.
- Second, the amount of processing needed to manage a table model which does not match the underlying representation could lead to unreasonable delays in transmitting changes to the network processor.

3.1 Unified Table Model

Implementations that utilize the unified table model to represent IPv6 unicast forwarding information use a single data entity, which shall be subsequently referred to as a "FIB Table."[1] This table is comprised of entries, each one consisting of a prefix and an array of next hop information.

To facilitate capabilities such as load balancing or ECMP, the next hop array may contain information for one or more next hops. Each next hop array is essentially a set of next hops. There are different flavors of next hop, each of which dictates a different forwarding action. The basic, direct attach, and remote types forward packets through the network processor. These forwarding next hop types have an IP destination address and an egress interface included in their definition. Other flavors of next hop indicate other actions such as discard the packet or forward the packet to the control plane.

Figure 3 illustrates the conceptual layout of a FIB table and several table entries. In this structure, the unique "key" used to reference an entry is the prefix element.

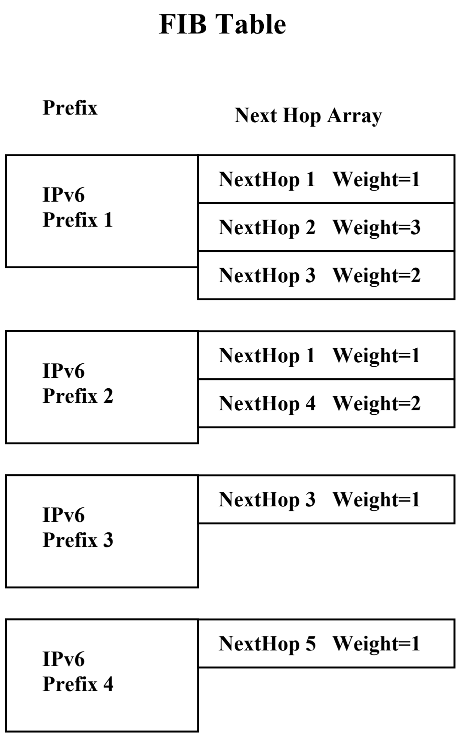

Figure 3 - Unified FIB Table Model

Address resolution in the unified table mode is modeled separately, using the distinct address resolution functions and data types described later in this specification.

[1] The term FIB is an acronym used for a Forwarding Information Base and is used throughout the document when referring to the forwarding information abstraction. However, note that the term "FIB Table" has been chosen to refer to the unified mode data entity used to model forwarding information. Therefore, it is used in the nomenclature of the unified mode data structures and API function names.

3.2 Discrete Table Model

Implementations that utilize the discrete table model to represent IPv6 unicast forwarding information use two separate data entities, which shall be subsequently referred to as the "Prefix Table" and the "Next Hop Table." The prefix table is comprised of entries, each one consisting of a prefix and a next hop identifier that uniquely indicates an entry in a next hop table. The next hop table is comprised of entries, each one consisting of a next hop identifier and an array of next hop information. As with the unified mode FIB table, the next hop array can contain one or more elements of next hop information.

In order to forward a packet, each IP destination address specified in the prefix must have one or more next hops associated with it. In the discrete model, this association is provided by the next hop identifier, which correlates a prefix table entry to an entry in the next hop table. This "split" table model provides several benefits in some system designs. For example, some classes of high-performance networking nodes (e.g. – BGP routers) require optimal FIB updates when a set of routes changes. With a discrete model implementation, it may be possible to efficiently update forwarding information by altering a subset of next hop table entries. Whereas, in a unified model, it may be required that a larger set of FIB table entries be modified to accomplish the same forwarding information update.

Figure 4 illustrates the conceptual layout and relationship of the prefix table and next hop table. In the prefix table, the unique "key" used to reference an entry is the prefix element. In the next hop table, the unique "key" used to reference an entry is the next hop identifier.

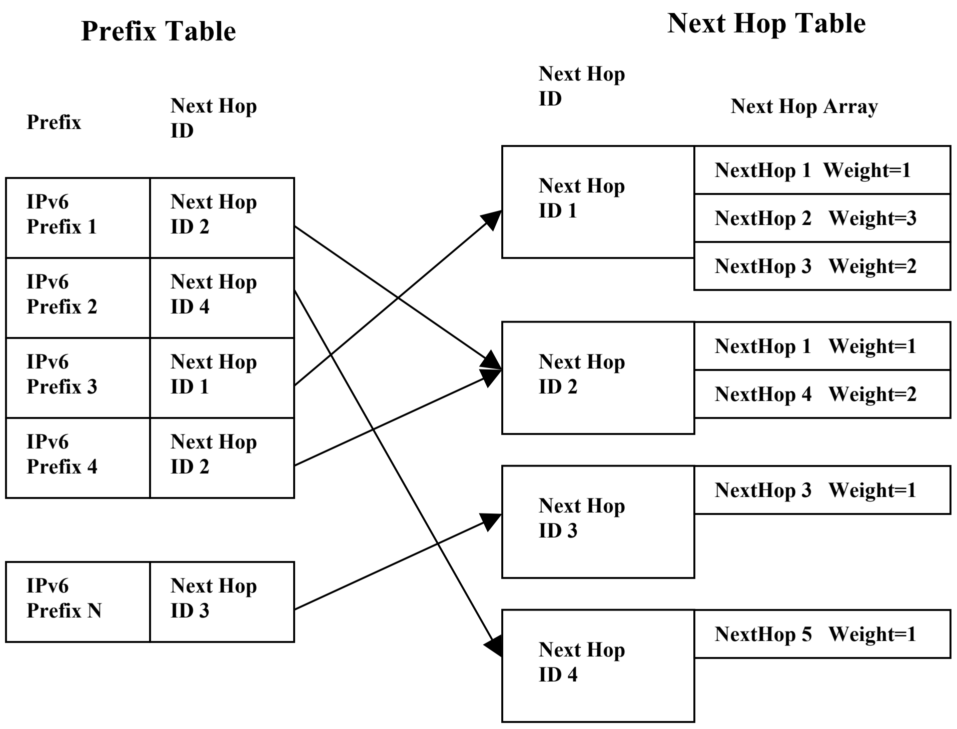

Figure 4 - Discrete Table Model

An application may create multiple prefix and next hop tables. The API defines a function that creates a relationship between a prefix table and a next hop table. Such relationships can be one-to-one, so that there is a prefix table corresponding to each next hop table. Additionally, the relationship may be many-to-one, in which two or more prefix tables share a single next hop table. A one-to-many relationship, in which a single prefix table is associated with multiple next hop tables, is not supported.

The application is responsible for the allocation and use of next hop identifier values. It may choose any values it wants, and it may re-use them in any way it wants. An application should create a next hop table entry for each new next hop identifier it uses in the prefix table. If an NP forwarding element references a prefix table entry containing a next hop identifier that is not assigned to a valid next hop table entry, the implementation may generate an event to signal the application of the problem.

Address resolution in the discrete table mode is also modeled separately, using the distinct address resolution functions and data types described later in this specification.

3.3 Address Resolution Table

For both the unified and discrete model, the next hop information in a next hop array element contains IP-level address and egress interface information. In order to forward a packet, the next hop IP address and egress interface must be resolved to a layer 2 address. The address resolution table makes this possible, by taking an IP address and egress interface as key fields, and providing a media specific address.

Neighbor Discovery may be implemented in several ways, e.g. by means of a Control Plane application. Alternatively, Neighbor Discovery could be implemented in the forwarding blades.

Neighbor Discovery interacts directly with data forwarding in two situations:

- When a packet should be forwarded and no entry exists in the Address Resolution Table for that nexthop. Neighbor Discovery must then resolve the nexthop address and install an entry in the Address Resolution Table.
- When a packet should be forwarded and the Address Resolution Table entry is in the *stale* state. The *stale* state indicates that the destination has been reachable and the packet can be sent, but a refresh is requested.

A reachability state is included in the Address Resolution Table entries for supporting exchange of state information with the NP forwarding element. The Neighbor Discovery Handler must maintain the state to enable the NP forwarding element to distinguish between a reachable entry and a stale entry.

Figure 5 gives an illustration of the model with the Neighbor Discovery Handler implemented in the Control Plane and using this SAPI.

In the Control Plane the reachability states are then used as defined by the Neighbor Discovery protocol:

incomplete	Address resolution is being performed on the entry. Specifically, a Neighbor Solicitation has been sent to the solicited-node multicast address of the target, but the corresponding Neighbor Advertisement has not yet been received.
reachable	Positive confirmation was received within the last ReachableTime milliseconds that the forward path to the neighbor was functioning properly. While *reachable*, no special action takes place as packets are sent.
stale	More than ReachableTime milliseconds have elapsed since the last positive confirmation was received that the forward path was functioning properly. While *stale*, no action takes place until a packet is sent. The *stale* state is also entered upon receiving an unsolicited Neighbor Discovery message that updates the cached link-layer address. Receipt of such a message does not confirm reachability, and entering the *stale* state insures reachability is verified quickly if the entry is actually being used. However, reachability is not actually verified until the entry is actually used.

delay	More than ReachableTime milliseconds have elapsed since the last positive confirmation was received that the forward path was functioning properly, and a packet was sent within the last DELAY_FIRST_PROBE_TIME seconds. If no reachability confirmation is received within DELAY_FIRST_PROBE_TIME seconds of entering the *delay* state, send a Neighbor Solicitation and change the state to *probe*. The *delay* state is an optimization that gives upper-layer protocols additional time to provide reachability confirmation in those cases where ReachableTime milliseconds have passed since the last confirmation due to lack of recent traffic. Without this optimization the opening of a TCP connection after a traffic lull would initiate probes even though the subsequent three-way handshake would provide a reachability confirmation almost immediately.
probe	A reachability confirmation is actively sought by retransmitting Neighbor Solicitations every RetransTimer milliseconds until a reachability confirmation is received.

Above

- ReachableTime is the time a neighbor is considered reachable after receiving a reachability confirmation
- RetransTimer is the time between retransmissions of Neighbor Solicitation messages to a neighbor when resolving the address or when probing the reachability of a neighbor
- DELAY_FIRST_PROBE is a Neighbor Discovery protocol constant, currently being 5 seconds.

In the Forwarding Plane the states are used as follows:

incomplete	No layer 2 address is available for the entry. A corresponding TRANSITION event has been sent across the SAPI. A small queue of packets, waiting for a layer 2 address to be available, is maintained.
reachable, *delay*, *probe*	A layer 2 address is available for the entry and may be used. No special action takes place as packets are sent. In the forwarding plane do not distinguish between the *reachable*, *delay* and *probe* states.
stale	While *stale*, a layer 2 address is available for the entry, however no special action takes place until a packet is sent. When a packet is sent, a corresponding TRANSITION event is sent across the SAPI and the *reachable*, *delay* or *probe* state is entered.

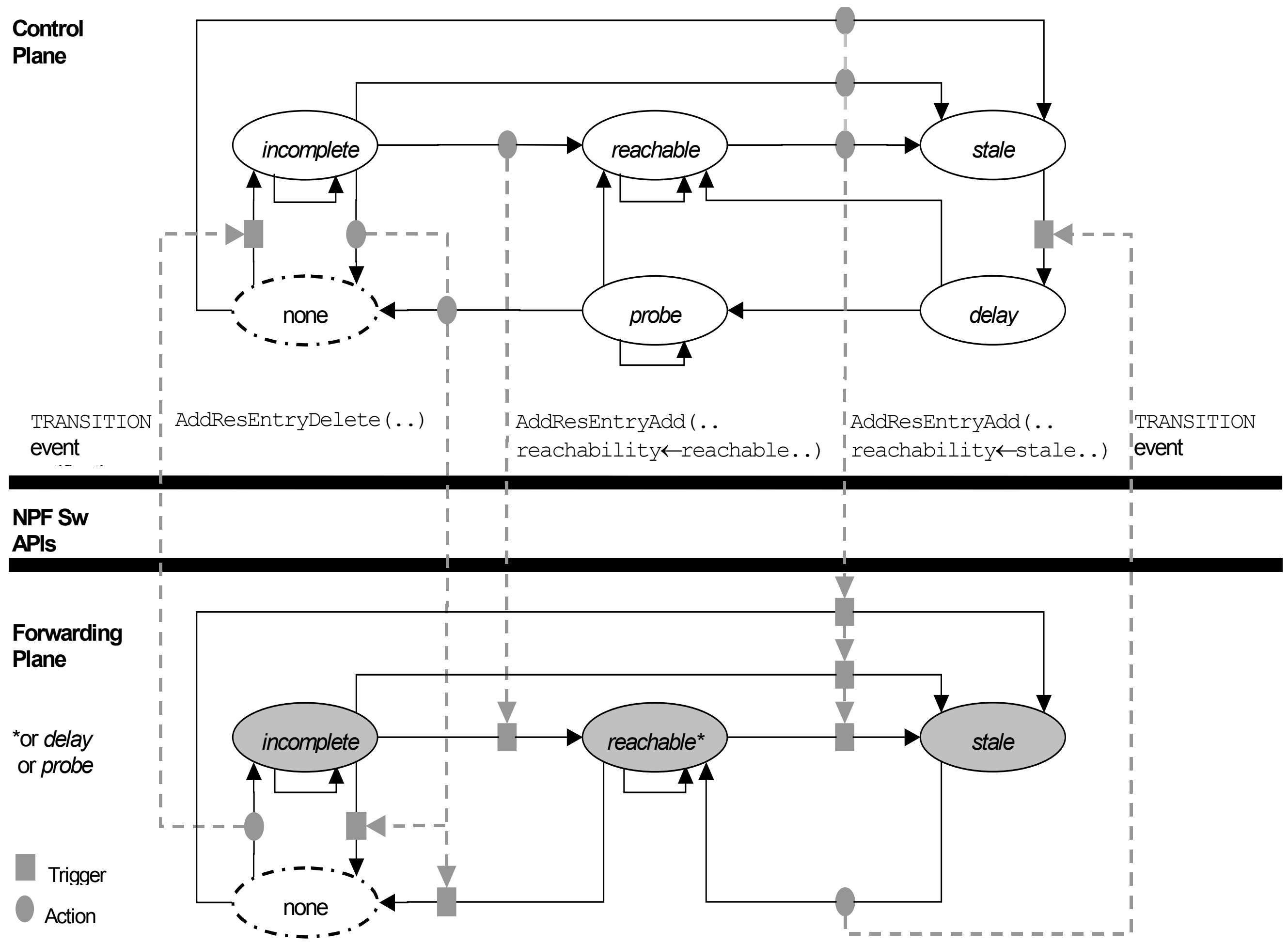

Figure 5 - IPv6 Address Resolution Model

Figure 5 is a simplified state diagram to illustrate the principles of the ND protocol. Additional transitions are present to cope with irregular behaviour, i.e. exchange of link address.In the Forwarding Plane, the forwarding of an IPv6 packet triggers, when no entry in the relevant Address Resolution Table exists for the next hop, a new entry in this Address Resolution Table in the *incomplete* state, as well as the queuing of the packet and the propagation of a corresponding TRANSITION event over the SAPI. This triggers the Neighbor Discovery handler to proceed with address resolution according to the Neighbor Discovery protocol. When the IPv6 address is resolved, the Neighbor Discovery handler informs this, by a SAPI request, to the Forwarding Plane, where it triggers a transition to the *reachable* state, and sending of queued packets. When the Neighbor Discovery handler changes the state of the corresponding entry to *stale* state, this is informed, by a SAPI request, to the Forwarding Plane, where it triggers a transition to *stale* state. The sending of the next relevant IPv6 packet triggers in the Forwarding Plane a state change back to the *reachable* (or *delay* or *probe*) state, as well as the propagation of a corresponding TRANSITION event over the SAPI, such that the Neighbor Discovery handler may initiate some validation of the reachability information in parallel with packet forwarding. Should the Neighbor Discovery handler delete an entry, this is informed to the Forwarding layer, by a SAPI request.

When Neighbor Discovery is implemented in forwarding blades, it is the SAPI implementation that uses the reachability states as defined by the Neighbor Discovery protocol. The SAPI implementation may use or ignore requests regarding address resolution, and application may use or ignore events regarding address resolution. As an example, upper layer reachability confirmation indications may be mapped to address resolution related update requests that may be used or ignored by the SAPI implementation.

An address resolution table entry contains media specific information for IP address and egress interface pairs. Because there are many types of physical media, the media address component is defined as a type field, indicating the media type, and a union of media addresses definitions.

Figure 6 illustrates the conceptual layout of the address resolution table. As mentioned, the unique "key" used to reference an entry is the combination of the IP address and the egress interface identifier.

Address Resolution Table

Next Hop IP Address	Interface Handle	Media Address	Reachability
IP Address 1	Interface 4	MAC Address A	Reachable
IP Address 2	Interface 3	MAC Address C	Reachable
IP Address 3	Interface 1	MAC Address X	Stale
IP Address 4	Interface 6	MAC Address D	Reachable

Figure 6 - Address Resolution Table Model - (Ethernet example)

To provide further flexibility, note that the four pieces of information that comprise an address resolution table entry are also defined in the next hop information in a next hop array element. This allows some implementations to avoid an additional address resolution table lookup because the media address is available immediately when the next hop is determined.

3.4 API Usage Guidelines

Application clients of the IPv6 Services API will create one or more instances of IPv6 tables to control the IPv6 forwarding of a device. In order to determine what type of table to create, implementations may use the query methods defined in section 5.5 to determine which table modes are supported and which mode provides the best performance. Once an application has determined the supported and preferred modes of operation, it may create one or more Prefix, Next Hop, FIB, and Address Resolution tables, using the functions associated with each table type to populate and monitor each table.

Some implementations may only support a unified table mode of operation. Unified table-only implementations will return an error code if discrete prefix or next hop table functions are invoked.

Unified and discrete table implementations will support the address resolution functions.

Implementations that support both discrete and unified table mode of operation may be used in both modes at the same time, however, it is assumed that each mode is used for a unique and distinct FIB. In other words, the two different modes should not be used to operate on the same FIB.

The unified FIB table functions may not be used with the discrete next hop and prefix table handles and discrete next hop and prefix table functions may not be used with unified FIB table handles. Type checking will detect such misuse at compile time, or, if not detected, the implementation will return errors when discrete handles are used with unified functions and vice versa.

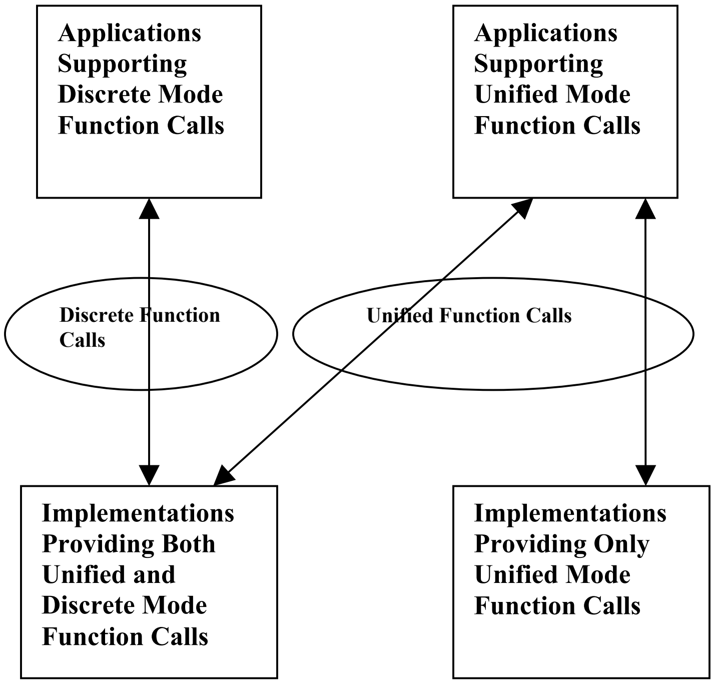
Applications Supporting Discrete Mode Function Calls
Applications Supporting Unified Mode Function Calls
Discrete Function Calls
Unified Function Calls
Implementations Providing Both Unified and Discrete Mode Function Calls
Implementations Providing Only Unified Mode Function Calls

Figure 7 - Usage Models

4 Data Types

This section defines data types that are used in the unified and discrete mode implementations as well as shared data types such as return codes, table mode query values, next hop information and IPv6 address resolution data structures. In addition, this section provides data structures used for asynchronous completion callbacks and event notifications.

4.1 Common Data Types

4.1.1 Table Mode Query Data Types

This section defines the types used by an application to query the supported and preferred table modes of an implementation.

```
typedef enum {
      NPF_IPV6UC_UNIFIED_ONLY       = 1,
      NPF_IPV6UC_BOTH_SUPPORTED     = 2
} NPF_IPv6UC_SupportedMode_t;

typedef enum {
      NPF_IPV6UC_DISCRETE_PREFERRED = 1,
      NPF_IPV6UC_UNIFIED_PREFERRED  = 2,
      NPF_IPV6UC_NO_PREFERENCE      = 3
} NPF_IPv6UC_PreferredMode_t;
```

4.1.2 Prefix Data Types

This simple data type provides a more meaningful name for the structure that defines an IPv6 address and prefix length.

```
/*
 * IPv6 prefix type
 */
typedef struct {
      NPF_IPv6Address_t   IPv6Addr;
      NPF_uint8_t         IPv6Plen;
} NPF_IPv6Prefix_t;

/*
 * IPv6 prefix retype
 */
typedef NPF_IPv6Prefix_t  NPF_IPv6UC_Prefix_t;
```

4.1.3 Next Hop Array Data Types

A Next Hop Table entry consists of a Next Hop identifier and a Next Hop array, whereas, a FIB Table entry consists of a prefix and a Next Hop array. In each case, there exists a Next Hop array which defines one or more next hops with a count to indicate the number of next hops in the array.

- nextHopCount – The number of next hops in the Next Hop array.
- nextHopArray – An array of next hops. If multiple next hops are specified, the weight member of the NPF_IPv6UC_NextHop_t structure is used to determine which data packets to forward to each next hop. The algorithm used to select particular next hops is implementation dependent.

```
/*
 * IPv6 unicast Next Hop Array: nextHopArray points to an array
 * (one or more) of NPF_IPv6UC_NextHop_t structures.
 * nextHopCount indicates how many next hops are in the array.
 */
typedef struct {
      NPF_uint32_t             nextHopCount;
      NPF_IPv6UC_NextHop_t    *nextHopArray;
} NPF_IPv6UC_NextHopArray_t;
```

Each member of the nextHopArray specifies a particular next hop definition, with the following components:

- type – The type of the next hop:
    - basic – The forwarding behavior for a basic entry is to send the packet to the Next Hop IP address in the next hop.
    - directAttach – A directly attached subnet means that the IPv6 destination address is on a directly attached network, and the Next Hop IP address is either absent or is identical to the IPv6 destination address. The forwarding behavior is modified by selecting a media address corresponding to the destination IP address, not a next hop router IP address. Support for this type is optional.
    - sendToCP – Forwarding behavior for this type is for the network processor to send the packet to a Control Plane. Other than type, no other fields are used for this kind of next hop.
    - discard – The network processor counts the packet and then drops it. Other than type, there are no other fields for this kind of next hop.
    - remote – The next hop IP address is an address of a remote router through which this packet will be forwarded, not the immediate next hop IP address. The egress interface handle MAY be absent. Route table entries with prefixes learned through the BGP protocol MAY use these Next Hop Entries. Forwarding behavior modification, if any, is the implementer's choice. One possible use of the Remote type is in optimization of prefix table updates on BGP routes with an IBGP switch over. Support for this type is optional.
    - tunnel – The Next Hop IP address is the address of the next hop in the tunnel. The egress interface handle points to a tunnel pseudo-interface. The Tunnel Exit Node address, which is the final destination of the encapsulating packet, is held in the interface structure.
- weight – Has meaning when the Next Hop array contains a list of multiple interchangeable next hops. One possible use may be for the forwarder to assign each packet to one next hop in the list, while trying to keep the link bandwidth utilization of each proportional to its own weight, relative to the rest of the list. This parameter can be used in various environments, such as link bundling, ECMP, traffic engineering and others. How this value is assigned and used is outside the scope of this document.
- egressInterface – The handle of the interface representing the egress path to which the next hop router is connected.
- nextHopIP – The IPv6 address of the next hop router or end system.
- mediaAddress – Optional media address associated with the next hop IP address. This parameter may, together with the reachability parameter below, be used to populate lower layer information in the FIB, if the FIB is organized to contain such information. However, some implementations hold lower layer information in a different table; the address resolution table. In such a case, the address resolution table function calls will provide the means to manipulate the lower layer information. This lower layer information may be provided by one means or the other[2], but in general, both methods

[2] An implementation MAY ignore L2 addresses from the IPv6 API, if it has a better source for the information, such as Neighbor Discovery directly implemented on a line card.

should not be used together. If an implementation does provide the means to use both this parameter and the address resolution table function calls, the preference of which function call to use is application dependent.

- reachability – Optional reachability state associated with the next hop IP address. This parameter may, together with the mediaAddress parameter above, be used to populate lower layer information in the FIB, as described for the mediaAddress parameter above.

```
/*
 * IPv6 unicast next hop structure: weight, egressInterface and
 * nextHopIP fields are valid only for NPF_IPV6UC_NH_BASIC,
 * NPF_IPV6UC_NH_DIRECT_ATTACH, NPF_IPV6UC_NH_REMOTE, and
 * NPF_IPV6UC_NH_TUNNEL types.
 */
typedef struct {
      NPF_IPv6UC_NextHopType_t      type;
      NPF_uint16_t                  weight;
      NPF_IfHandle_t                egressInterface;
      NPF_IPv6Address_t             nextHopIP;
      NPF_MediaAddress_t            mediaAddress;
      NPF_IPv6_Reachability_t       reachability;
} NPF_IPv6UC_NextHop_t;

typedef enum {
      NPF_IPV6UC_NH_BASIC         =  1,
      NPF_IPV6UC_NH_DIRECT_ATTACH =  2,
      NPF_IPV6UC_NH_SEND_TO_CP    =  3,
      NPF_IPV6UC_NH_DISCARD       =  4,
      NPF_IPV6UC_NH_REMOTE        =  5,
      NPF_IPV6UC_NH_TUNNEL        =  6
} NPF_IPv6UC_NextHopType_t;
```

4.1.4 Address Resolution Data Types

This section defines the IPv6 control structures that are required to perform IPv6 address to physical address resolution. IPv6 address to physical address resolution is performed by the IPv6 Neighbor Discovery protocol as described by RFC 2461.

An Address Resolution Table entry consists of an IP address, an interface handle, media specific address and a reachability state of the entry. These components are defined in a single NPF_IPv6UC_AddResEntry_t structure.

- IP_Address - The protocol address for which a media-specific address binding is defined.
- interfaceHandle - The interface which is associated with this entry.
- mediaAddress - The media address associated with the specified protocol address. Examples might be a 6 byte Ethernet MAC address or an ATM VPI/VCI.
- reachability - The reachability state of the entry.

```
/*
IPv6 unicast Address Resolution entry:
 */
typedef struct {
      NPF_IPv6Address_t       IP_Address;
      NPF_IfHandle_t          interfaceHandle;
      NPF_MediaAddress_t      mediaAddress;
      NPF_IPv6_Reachability_t reachability;
} NPF_IPv6UC_AddResEntry_t;
```

When performing a query of the address resolution table, the NPF_IPv6UC_AddResKey_t structure defines the search key.

```
/*
 * This structure contains the key of an Address Resolution
 * table entry, consisting of IP address and interface handle.
 */
typedef struct {
      NPF_IPv6Address_t IP_Address;
      NPF_IfHandle_t    interfaceHandle;
} NPF_IPv6UC_AddResKey_t;
```

For an Address Resolution Table query, the response structure is identical to the NPF_IPv6UC_AddResEntry_t structure defined above. Instead of duplicating this structure with a unique name, a simple typedef is defined and this structure name is then used in the completion callback asynchronous response union.

```
/*
 * This simple data type provides a more meaningful name for the
 * structure used in the address resolution asynchronous callback data.
 */
typedef  NPF_IPv6UC_AddResEntry_t    NPF_IPv6UC_AddResQueryResp_t;
```

The media specific address structure is defined in a common NPF header file since it is used by several APIs (including the NPF Software Implementation Agreement – IPv4 Unicast Forwarding Service API). It is replicated here as a comment for informative purposes.

The media specific address structure contains a type field to identify the particular format.

- type – The type of address contained in the mediaAddress parameter.

```
/*
 * Media Address structure:
 */
typedef struct {
      NPF_MediaType_t          type;
      union {
             NPF_MAC_Address_t MAC_Address;
             NPF_VccAddr_t     ATM_Vc;
      }u;
} NPF_MediaAddress_t;

/*
 * Media type definition:
 */
typedef  enum {
      NPF_NO_MEDIA_TYPE =  1,
      NPF_MAC_ADDRESS   =  2,
      NPF_ATM_VC       =  3
} NPF_MediaType_t;
```

The IPv6 address resolution reachability type is defined as:

```
typedef  enum {
      NPF_IPv6_VOID           =  0,
      NPF_IPv6_NONE           =  1,
      NPF_IPv6_INCOMPLETE      =  2,
      NPF_IPv6_REACHABLE       =  3,
      NPF_IPv6_STALE           =  4,
      NPF_IPv6_DELAY           =  5,
      NPF_IPv6_PROBE           =  6
} NPF_IPv6_Reachability_t;
```

where the reachability constants means:

- `NPF_IPv6_VOID`: no address resolution reachability state information present
- `NPF_IPv6_NONE`: address resolution entry, required to send packet, is missing
- `NPF_IPv6_INCOMPLETE`: address resolution is in progress, link-layer addr. not determined
- `NPF_IPv6_REACHABLE`: link-layer addr. determined, neighbor known reachable recently
- `NPF_IPv6_STALE`: neighbor no longer known reachable recently, no traffic sent yet
- `NPF_IPv6_DELAY`: traffic sent to neighbor, delay sending probes for a short while
- `NPF_IPv6_PROBE`: unicast Neighbor Solicitation probes being sent to verify reachability.

4.1.5 Table Types

The Address Resolution Table id is a nonzero integer value assigned by the application to each Address Resolution Table. No two Address Resolution Tables may have the same Address Resolution Table id value. The Address Resolution Table id performs at least two functions: it serves to identify the Address Resolution Tables in callbacks, and it aids in recovering from the unlikely event of a lost callback from the Address Resolution Table creation function. The IPv6 Unicast Forwarding Service API implementation must remember the Address Resolution Table id value associated with each Address Resolution Table Handle it creates. Any attempt by the application to create a new Address Resolution Table Handle using an Address Resolution Table id value already associated with an existing handle must result in an error being returned to the application, and no new handle created. Callback information from functions that create, modify, destroy or query Address Resolution Tables must always include the Address Resolution Table id value for each Address Resolution Table referenced.

```
typedef NPF_uint32_t NPF_IPv6UC_AddResTableId_t;
```

An Address Resolution Table is uniquely identified by a table handle which is defined as follows:

```
typedef NPF_uint32_t NPF_IPv6UC_AddResTableHandle_t;
```

The IPv6 Unicast Forwarding API supports forwarding tables of two types, unified and discrete, which each have their own corresponding handle types. Other APIs, such as Interface Management and Packet Handler, depend on the IPv6 Unicast Forwarding API for handle definitions because they contain functions that refer to forwarding tables. The following Forwarding Table Handle represents a forwarding table (FIB) regardless of its mode. It is used in other APIs, so as not to expose the table's type (unified or discrete) outside of the IPv6 Unicast Forwarding API.

```
typedef NPF_uint32_t NPF_IPv6UC_FwdTableHandle_t;
```

4.1.6 Return Codes

This section defines IPv6 Unicast Forwarding API return codes that are used for IPv6 forwarding and address resolution function calls. These codes are used for returns from asynchronous API function calls.

```
/*
 * Asynchronous error codes (returned in function callbacks)
 */

typedef NPF_uint32_t NPF_IPv6UC_ReturnCode_t;

#define IPV6_ERR(n) ((NPF_IPv6UC_ReturnCode_t) NPF_IPV6_BASE_ERR + (n))

#define NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST         IPV6_ERR(1)
#define NPF_IPV6UC_E_INVALID_HANDLE                     IPV6_ERR(3)
#define NPF_IPV6UC_E_INSUFFICIENT_STORAGE               IPV6_ERR(4)
```

4.2 Unified Mode Data Types

4.2.1 FIB Table Query Data Type

This section defines the IPv6 specific control structures used for querying FIB table contents in unified implementations. The asynchronous callback data will contain one or more of the following NPF_IPv6UC_FibQueryResp_t structures. More than one structure is provided if multiple FIB table entries are queried at once.

- prefix – The prefix used to locate a particular entry in the FIB Table.
- nextHopArray – The set of one or more next hop definitions related to this prefix.

```
/*
 * This structure contains the query results for a single FIB
 * table entry.
 */
typedef struct {
      NPF_IPv6UC_Prefix_t           prefix;
      NPF_IPv6UC_NextHopArray_t     nextHopArray;
} NPF_IPv6UC_FibQueryResp_t;
```

4.2.2 Table Types

The FIB Table id is a nonzero integer value assigned by the application to each FIB Table. No two FIB Tables may have the same FIB Table id value. The FIB Table id performs at least two functions: it serves to identify the FIB Tables in callbacks, and it aids in recovering from the unlikely event of a lost callback from the FIB Table creation function. The IPv6 Unicast Forwarding Service API implementation must remember the FIB Table id value associated with each FIB Table Handle it creates. Any attempt by the application to create a new FIB Table Handle using an FIB Table id value already associated with an existing handle must result in an error being returned to the application, and no new handle created. Callback information from functions that create, modify, destroy or query FIB Tables must always include the FIB Table id value for each FIB Table referenced.

```
typedef NPF_uint32_t NPF_IPv6UC_FibTableId_t;
```

A FIB Table is uniquely identified within the scope of the IPv6 Unicast Forwarding API by a table handle which is defined as follows:

```
typedef NPF_uint32_t NPF_IPv6UC_FibTableHandle_t;
```

Note that external to the IPv6 Unicast Forwarding API, a FIB table is uniquely identified by a table handle which is defined by the data type NPF_IPv6UC_FwdTableHandle_t. This specification could have forced API calls external to the IPv6 Unicast Forwarding API to specify a "FIB type" parameter and the internal FIB handle type. However, for a cleaner interface, a decision was made to use a single external handle type to identify a FIB, regardless of how the IPv6 function was structuring the information.

The following structure is used in the callback from the FIB Table handle creation function, to return two handles. The internal handle is used within the scope of the IPv6 unicast forwarding API and the external handle is used by other NPF API's when referencing an IPv6 FIB. It is the responsibility of the implementation to maintain the mapping between these two handles, which refer to the same FIB.

```
/*
 * Async Response struct for NPF_IPv6UC_FIBTableHandleCreate()
 */
typedef struct {
      NPF_IPv6UC_FwdTableHandle_t    extHandle;
      NPF_IPv6UC_FibTableHandle_t    intHandle;
} NPF_IPv6UC_FibCreateResp_t;
```

4.3 Discrete Mode Data Types

4.3.1 Prefix Table Query Data Type

This section defines the IPv6 specific control structures used for querying prefix table contents in discrete implementations. The asynchronous callback data will contain one or more of the following NPF_IPv6UC_PrefixQueryResp_t structures. More than one structure is provided if multiple prefix table entries are queried at once.

- prefix – The prefix used to locate a particular entry in the Prefix Table.
- nextHopIdentifier – The identifier of the next hop array in the Next Hop Table for this particular prefix.

```
/*
 * This structure contains the query results for a single prefix
 * table entry.
 */
typedef struct {
      NPF_IPv6UC_Prefix_t      prefix;
      NPF_uint32_t             nextHopIdentifier;
} NPF_IPv6UC_PrefixQueryResp_t;
```

4.3.2 Next Hop Table Query Data Type

This section defines the IPv6 specific control structures used for querying next hop table contents in discrete implementations. The asynchronous callback data will contain one or more of the following NPF_IPv6UC_NextHopQueryResp_t structures. More than one structure is provided if multiple next hop table entries are queried at once.

- nextHopIdentifier – The next hop identifier used to locate a particular entry in the Next Hop Table.
- nextHopArray – The set of one or more next hop definitions related to this particular next hop identifier.

```
/*
 * This structure contains the query results for a single next
 * hop table entry.
 */
typedef struct {
      NPF_uint32                   nextHopIdentifier;
      NPF_IPv6UC_NextHopArray_t    nextHopArray;
} NPF_IPv6UC_NextHopQueryResp_t;
```

4.3.3 Table Types

The Prefix Table id is a nonzero integer value assigned by the application to each Prefix Table. No two Prefix Tables may have the same Prefix Table id value. The Prefix Table id performs at least two functions: it serves to identify the Prefix Tables in callbacks, and it aids in recovering from the unlikely event of a lost callback from the Prefix Table creation function. The IPv6 Unicast Forwarding Service API implementation must remember the Prefix Table id value associated with each Prefix Table Handle it creates. Any attempt by the application to create a new Prefix Table Handle using a Prefix Table id value already associated with an existing handle must result in an error being returned to the application, and no new handle created. Callback information from functions that create, modify, destroy or query Prefix Tables must always include the Prefix Table id value for each Prefix Table referenced.

```
typedef NPF_uint32_t NPF_IPv6UC_PrefixTableId_t;
```

A Prefix Table is uniquely identified within the scope of the IPv6 Unicast Forwarding API by a table handle, which is defined as follows:

```
typedef NPF_uint32_t NPF_IPv6UC_PrefixTableHandle_t;
```

Note that external to the IPv6 Unicast Forwarding API, a prefix table is uniquely identified by a table handle which is defined by the data type NPF_IPv6UC_FwdTableHandle_t. This specification could have forced API calls external to the IPv6 Unicast Forwarding API to specify a "FIB type" parameter and the internal FIB handle type. However, for a cleaner interface, a decision was made to use a single external handle type to identify a FIB, regardless of how the IPv6 function was structuring the information.

The following structure is used in the callback from the Prefix Table handle creation function, to return two handles. The internal handle is used within the scope of the IPv6 Unicast forwarding API and the external handle is used by other NPF API's when referencing an IPv6 FIB. It is the responsibility of the application to maintain the mapping between these two handles, which refer to the same FIB.

```
/*
 * Async Response struct for NPF_IPv6UC_PrefixTableHandleCreate()
 */
typedef struct {
      NPF_IPv6UC_FwdTableHandle_t    extHandle;
      NPF_IPv6UC_PrefixTableHandle_t intHandle;
} NPF_IPv6UC_PfxCreateResp_t;
```

The Next Hop Table id is a nonzero integer value assigned by the application to each Next Hop Table. No two Next Hop Tables may have the same Next Hop Table id value. The Next Hop Table id performs at least two functions: it serves to identify the Next Hop Tables in callbacks, and it aids in recovering from the unlikely event of a lost callback from the Next Hop Table creation function. The IPv6 Unicast Forwarding Service API implementation must remember the Next Hop Table id value associated with each Next Hop Table Handle it creates. Any attempt by the application to create a new Next Hop Table Handle using a Next Hop Table id value already associated with an existing handle must result in an error being returned to the application, and no new handle created. Callback information from functions that create, modify, destroy or query Next Hop Tables must always include the Next Hop Table id value for each Next Hop Table referenced.

```
typedef NPF_uint32_t NPF_IPv6UC_NextHopTableId_t;
```

A Next Hop Table is uniquely identified by a table handle which is defined as follows:

```
typedef NPF_uint32_t NPF_IPv6UC_NextHopTableHandle_t;
```

4.4 Data Structures for Completion Callbacks

This section defines the control structures needed for a Completion Callback, which provides the response information to the application which invoked an asynchronous function call. Although an asynchronous function call can request the execution of a single operation, many functions can also request the execution of multiple operations. For example, an NPF_IPv6UC_AddResEntryAdd function call may choose to add a single address resolution entry to the Address Resolution table, but it may also add multiple entries with a single function call invocation.

When a single operation is requested, a single completion callback will occur. However, when multiple operations are requested, not all of these requests may complete at the same time. The implementation MAY invoke the completion callback one or more times in order to provide responses for the total number of operations requested. For this reason, the callback data structure is designed to be flexible in how it provides status on these responses and it also allows the bundling of one or more responses into a single callback invocation.

Each completion callback provides the NPF_IPv6UC_CallbackData_t structure, whose members will have particular values depending on the invoking function, whether or not a single operation was requested and whether the operations were successful or not. The following sections provide details regarding the data structures involved in a completion callback.

4.4.1 Completion Callback Structures

The NPF_IPv6UC_CallbackData_t structure is provided as a parameter when the callback function is invoked. The basic definition of the fields is provided below, while more detailed descriptions follow.

- type – This field indicates which function invocation led to this response.
- allOK – This field and the numResp field provide a flexible means of providing information regarding the number of responses in this callback and their status. The specific details for these fields are provided below.
- numResp – This field and the allOK field provide a flexible means of providing information regarding the number of responses in this callback and their status. The specific details for these fields are provided below
- resp – A pointer to an array of response elements or the NULL pointer. Each array element contains a return code, indicating the completion status of the request element, and possibly may contain other information specific to the type of request.

```
typedef struct {
      NPF_IPv6UC_CallbackType_t   type;
      NPF_boolean_t               allOK;
      NPF_uint32_t                numResp;
      NPF_IPv6UC_AsyncResponse_t *resp;
} NPF_IPv6UC_CallbackData_t;
```

The following section provides detailed information regarding the content and meaning of the members in the NPF_IPv6UC_CallbackData_t structure. There are several possibilities to consider.

The application invokes a function requesting a single operation:

- If allOK = TRUE, then numResp = 0 and the “resp” pointer is NULL. This indicates the operation completed successfully and there is no other additional response data to return.
- If allOK = FALSE, then numResp = 1 and the “resp” pointer points to a response structure. If the returnCode field indicates NPF_NO_ERROR, the operation completed

successfully and there is additional response data in the structure. Otherwise, the operation failed and the reason is indicated by the returnCode.

The application invokes a function requesting multiple operations:

- If all operations completed successfully at the same time and there is no additional response data to provide, then allOK = TRUE, numResp = 0 and the "resp" pointer is NULL.
- If all operations completed successfully at the same time, but there is additional response data to provide, then allOK = FALSE, numResp indicates the total number of requested operations and the "resp" pointer points to an array of response structures. The returnCode field will indicate NPF_NO_ERROR.
- If some operations completed, but not all, then:
    - allOK = FALSE, numResp = the number of request operations completed.
    - The "resp" pointer will point to an array of response structures, each one containing one element for each completed request. For operations that completed successfully, the returnCode field will indicate NPF_NO_ERROR and additional response data may be present, depending on the type of function invocation. For operations that failed, the reason is indicated by the returnCode field.

  Callback function invocations are repeated in this fashion until all requests are complete. Responses are not repeated for request elements already indicated as complete in earlier callback function invocations.

The NPF_IPv6UC_AsyncResponse_t data structure contains a return code indicating an error or the success of a particular request operation. The structure may also contain other optional information that was requested by the operation or the information may assist in correlating the response to the corresponding request operation when multiple operations are requested by the application.

For IPv6 asynchronous function invocations that operate upon a particular table, it is the responsibility of the invoking application to associate the table handle with the subsequent asynchronous response. It is suggested that the "correlator", supplied as an invocation parameter, be used for this purpose. For example, when invoking the NPF_IPv6UC_PrefixEntryAdd() function, the application can choose a correlator value that uniquely identifies, or points to, its own structures representing the forwarding table. This value returned by the implementation in each callback invocation can help the application know to which table the callback belongs. When the asynchronous NPF_IPv6UC_CallbackFunc() is called, the prefix table handle will be returned in its correlator parameter.

The type field in the NPF_IPv6UC_CallbackData_t structure identifies the asynchronous function call which has led to this callback, and therefore, the relevant member of the union.

One or more of the following structures may be provided to the callback function in the response array within the NPF_IPv6UC_CallbackData_t structure.

```
typedef struct {
      NPF_IPv6UC_ReturnCode_t                returnCode;
      union {
             NPF_IPv6UC_PrefixTableId_t      prefixTableId;
             NPF_IPv6UC_NextHopTableId_t     nextHopTableId;
             NPF_IPv6UC_FibTableId_t         fibTableId;
             NPF_IPv6UC_AddResTableId_t      addResTableId;
             NPF_uint32_t                    unused;
      } u1;
      union {
             NPF_IPv6UC_PfxCreateResp_t      prefixTableHandles;
             NPF_IPv6UC_Prefix_t             prefix
             NPF_IPv6UC_PrefixQueryResp_t    prefixQueryResult;
             NPF_IPv6UC_NextHopTableHandle_t nextHopTableHandle;
             NPF_uint32_t                    nextHopIdentifier;
             NPF_IPv6UC_NextHopQueryResp_t   nextHopQueryResult;
             NPF_IPv6UC_FibCreateResp_t      fibTableHandles;
             NPF_IPv6UC_Prefix_t             fibPrefix;
             NPF_IPv6UC_FibQueryResp_t       fibQueryResult;
             NPF_IPv6UC_AddResTableHandle_t  addResTableHandle;
             NPF_IPv6UC_AddResKey_t          addResKey;
             NPF_IPv6UC_AddResQueryResp_t    addResQueryResult;
             NPF_uint32_t                    tableSpaceRemaining;
             NPF_uint32_t                    unused;
      } u2;
} NPF_IPv6UC_AsyncResponse_t;
```

The following structure defines the completion callback type values.

```
/*
 * Common callback definition:
 */
typedef enum NPF_IPv6UC_CallbackType {
      NPF_IPV6UC_PREFIX_TABLE_HANDLE_CREATE        = 1,
      NPF_IPV6UC_PREFIX_TABLE_HANDLE_DELETE        = 2,
      NPF_IPV6UC_PREFIX_ENTRY_ADD                  = 3,
      NPF_IPV6UC_PREFIX_ENTRY_DELETE               = 4,
      NPF_IPV6UC_PREFIX_TABLE_FLUSH                = 5,
      NPF_IPV6UC_PREFIX_TABLE_ATTRIBUTE_QUERY      = 6,
      NPF_IPV6UC_PREFIX_ENTRY_QUERY                = 7,
      NPF_IPV6UC_PREFIX_NEXT_HOP_TABLE_BIND        = 8,
      NPF_IPV6UC_NEXT_HOP_TABLE_HANDLE_CREATE      = 9,
      NPF_IPV6UC_NEXT_HOP_TABLE_HANDLE_DELETE      = 10,
      NPF_IPV6UC_NEXT_HOP_ENTRY_ADD                = 11,
      NPF_IPV6UC_NEXT_HOP_ENTRY_DELETE             = 12,
      NPF_IPV6UC_NEXT_HOP_TABLE_FLUSH              = 13,
      NPF_IPV6UC_NEXT_HOP_TABLE_ATTRIBUTE_QUERY    = 14,
      NPF_IPV6UC_NEXT_HOP_ENTRY_QUERY              = 15,
      NPF_IPV6UC_FIB_TABLE_HANDLE_CREATE           = 16,
      NPF_IPV6UC_FIB_TABLE_HANDLE_DELETE           = 17,
      NPF_IPV6UC_FIB_ENTRY_ADD                     = 18,
      NPF_IPV6UC_FIB_ENTRY_DELETE                  = 19,
      NPF_IPV6UC_FIB_TABLE_FLUSH                   = 20,
      NPF_IPV6UC_FIB_TABLE_ATTRIBUTE_QUERY         = 21,
      NPF_IPV6UC_FIB_ENTRY_QUERY                   = 22,
      NPF_IPV6UC_ADDRESS_RES_TABLE_HANDLE_CREATE   = 23,
      NPF_IPV6UC_ADDRESS_RES_TABLE_HANDLE_DELETE   = 24,
      NPF_IPV6UC_ADDRESS_RES_ENTRY_ADD             = 25,
      NPF_IPV6UC_ADDRESS_RES_ENTRY_DELETE          = 26,
      NPF_IPV6UC_ADDRESS_RES_TABLE_FLUSH           = 27,
      NPF_IPv6UC_ADDRESS_RES_TABLE_ATTRIBUTE_QUERY = 28,
      NPF_IPV6UC_ADDRESS_RES_ENTRY_QUERY           = 29
} NPF_IPv6UC_CallbackType_t;
```

Function Name	**Type Code**	**Union Structure (u2)**
PrefixTableHandleCreate	PREFIX_TABLE_HANDLE_CREATE	prefixTableHandles
PrefixTableHandleDelete	PREFIX_TABLE_HANDLE_DELETE	unused
PrefixEntryAdd	PREFIX_ENTRY_ADD	prefix
PrefixEntryDelete	PREFIX_ENTRY_DELETE	prefix
PrefixTableFlush	PREFIX_TABLE_FLUSH	unused
PrefixTableAttributeQuery	PREFIX_TABLE_ATTRIBUTE_QUERY	tableSpaceRemaining
PrefixEntryQuery	PREFIX_ENTRY_QUERY	prefixQueryResult
PrefixNextHopTableBind	PREFIX_NEXT_HOP_TABLE_BIND	unused
NextHopTableHandleCreate	NEXT_HOP_TABLE_HANDLE_CREATE	nextHopTableHandle
NextHopTableHandleDelete	NEXT_HOP_TABLE_HANDLE_DELETE	unused
NextHopEntryAdd	NEXT_HOP_ENTRY_ADD	nextHopIdentifier
NextHopEntryDelete	NEXT_HOP_ENTRY_DELETE	nextHopIdentifier
NextHopTableFlush	NEXT_HOP_TABLE_FLUSH	unused
NextHopTableAttributeQuery	NEXT_HOP_TABLE_ATTRIBUTE_QUERY	tableSpaceRemaining
NextHopEntryQuery	NEXT_HOP_ENTRY_QUERY	nextHopQueryResult
FibTableHandleCreate	FIB_TABLE_HANDLE_CREATE	fibTableHandles
FibTableHandleDelete	FIB_TABLE_HANDLE_DELETE	unused
FibEntryAdd	FIB_ENTRY_ADD	fibPrefix
FibEntryDelete	FIB_ENTRY_DELETE	fibPrefix
FibTableFlush	FIB_TABLE_FLUSH	unused
FibTableAttributeQuery	FIB_TABLE_ATTRIBUTE_QUERY	tableSpaceRemaining
FibEntryQuery	FIB_ENTRY_QUERY	fibQueryResult
AddResTableHandleCreate	ADDRESS_RES_TABLE_HANDLE_CREATE	addResTableHandle
AddResTableHandleDelete	ADDRESS_RES_TABLE_HANDLE_DELETE	unused
AddResEntryAdd	ADDRESS_RES_ENTRY_ADD	addResKey
AddResEntryDelete	ADDRESS_RES_ENTRY_DELETE	addResKey
AddResTableFlush	ADDRESS_RES_TABLE_FLUSH	unused
AddResAttributeQuery	ADDRESS_RES_TABLE_ATTRIBUTE_QUERY	tableSpaceRemaining
AddResEntryQuery	ADDRESS_RES_ENTRY_QUERY	addResQueryResult

4.5 Data Structures for Event Notification

The following sections detail the information related to IPv6 Unicast events. When an event notification routine is invoked, one of the parameters will be a structure of information related to one or more events.

4.5.1 Event Notification Types

The event type indicates the type of event data in the union of event structures returned in NPF_IPv6UC_EventData_t.

```
/*
 * This structure enumerates the events defined for IPv6
 * Unicast forwarding.
 */
typedef enum NPF_IPv6UC_Event {
      NPF_IPV6UC_PREFIX_TBL_MISS        =  1,
      NPF_IPV6UC_NEXT_HOP_TBL_MISS      =  2,
      NPF_IPV6UC_FIB_PREFIX_MISS        =  4,
      NPF_IPV6UC_FWDTBL_REFRESH         =  5,
      NPF_IPV6UC_ADD_RES_TRANSITION     =  6
} NPF_IPv6UC_Event_t;
```

4.5.2 Event Notification Structures

This section describes the various events which MAY be implemented.

It is important to note that even if an implementation does not support any of these events, the implementation still needs to provide the register and deregister event function to enable interoperability.

Note that some of the event structures provide an internal handle identifying a FIB. It is the responsibility of the application to provide the mapping between these internal handles and any external FIB handles used in API invocations other than the IPv6 Unicast Forwarding API.

This structure defines all the possible event definitions for IPv6 Unicast. An event type field indicates which member of the two unions are relevant in the specific structure.

```
/*
 * This structure represents a single event in the event array. The
 * type field indicates the specific event in the union.
*/
typedef struct {
      NPF_IPv6UC_Event_t                   type;
      union {
             NPF_IPv6UC_PrefixTableId_t      prefixTableId;
             NPF_IPv6UC_NextHopTableId_t     nextHopTableId;
             NPF_IPv6UC_FibTableId_t         fibTableId;
             NPF_IPv6UC_AddResTableId_t      addResTableId;
             NPF_uint32_t                    unused;
      } u1;
      union {
             NPF_IPv6UC_PrefixTblMiss_t      prefixTblMiss;
             NPF_IPv6UC_NextHopTblMiss_t     nextHopTblMiss;
             NPF_IPv6UC_FIB_PrefixMiss_t     fibPrefixMiss;
             NPF_IPv6UC_FwdTbl_Refresh_t     fwdTableRefreshRequest;
             NPF_IPv6UC_AddResTransition_t   addResTransition;
      } u2;
} NPF_IPv6UC_EventData_t;
```

This event is triggered when the forwarding plane is unable to find a prefix table entry for a specific IP address. This event is optional.

```
/*
 * This event data identifies the prefix table and the destination
 * IP address that was not located during a lookup.
 */
typedef struct {
      NPF_IPv6UC_PrefixTableHandle_t   pfxTableHandle;
      NPF_IPv6Address_t                destIP_Address;
} NPF_IPv6UC_PrefixTblMiss_t;
```

This event is triggered when the forwarding plane is unable to find a next hop table entry for a specific next hop identifier. This event is optional.

```
/*
 * This event data identifies the next hop table and the next hop
 * identifier that was not located during a lookup.
 */
typedef struct {
      NPF_IPv6UC_NextHopTableHandle _t   nextHopTableHandle;
      NPF_uint32_t                       nextHopIdentifier;
} NPF_IPv6UC_NextHopTblMiss_t;
```

This event is triggered when the forwarding plane is unable to find a FIB table entry for a specific IP address. This event is optional

```
/*
 * This event data identifies the FIB table and the destination
 * IP address that was not located during a lookup.
 */

typedef struct {
      NPF_IPv6UC_FibTableHandle_t         fibTableHandle;
      NPF_IPv6Address_t                   destIP_Address;
} NPF_IPv6UC_FIB_PrefixMiss_t;
```

This event is triggered when the application or the IPv6 API implementation needs to be notified that a FIB needs to be refreshed on the forwarding plane. This event is optional.

```
/*
 * This event data identifies the unified or discrete table handle
 * identifying the FIB.
 */
typedef struct {
      NPF_IPv6UC_TableType_t                tableHandleType;
      union {
             NPF_IPv6UC_FibTableHandle_t    fibTableHandle;
             NPF_IPv6UC_PrefixTableHandle_t prefixTableHandle;
      } u;
} NPF_IPv6UC_FwdTbl_Refresh_t;

/*
 * This structure defines the enumerations for the table type used in
 * the NPF_IPv6UC_FwdTbl_Refresh_t structure above.
 */

typedef enum NPF_IPv6UC_TableType {
      NPF_IPV6UC_FIB_TABLE       = 1,
      NPF_IPV6UC_PREFIX_TABLE    = 2
}NPF_IPv6UC_TableType_t;
```

This event is triggered when the forwarding plane performs a transition form one state of an address resolution entry to another state. This event is optional.

```
/*
 * This event data identifies the table and the address resolution
 * information as defined immediately after the corresponding
 * transition has been performed, and furthermore includes the
 * previous reachability state as defined immediately before the
 * transition
 */
typedef struct {
      NPF_IPv6UC_AddResTableType_t            addResTableType;
      union {
             NPF_IPv6UC_AddResTableHandle_t   addResTableHandle;
             NPF_IPv6UC_FibTableHandle_t      fibTableHandle;
             NPF_IPv6UC_NextHopTableHandle_t  nextHopTableHandle;
            } u;
      NPF_IPv6UC_AddResEntry_t                addResEntry;
      NPF_IPv6_Reachability_t                 previousReachability;
} NPF_IPv6UC_AddResTransition_t;

/*
 * This structure defines the enumerations for the table type used in
 * the NPF_IPv6UC_AddResTransition_t structure above.
 */

typedef enum NPF_IPv6UC_AddResTableType {
      NPF_IPV6UC_ADD_RES_ADDRES_TABLE     = 1,
      NPF_IPV6UC_ADD_RES_FIB_TABLE        = 2,
      NPF_IPV6UC_ADD_RES_NEXTHOP_TABLE    = 3
}NPF_IPv6UC_AddResTableType_t;
```

This structure represents the data parameter provided when the event notification routine is invoked. It contains a count of events and an array of structures providing event specific information.

```
/*
 * This structure is provided when the event notification handler
 * is invoked. It specifies one or more IPv6 Unicast Forwarding events.
 */
typedef struct {
      NPF_uint32_t                 numEvents;
      NPF_IPv6UC_EventData_t      *eventArray;
} NPF_IPv6UC_EventArray_t;
```

4.5.3 NPF_eventMask Bit Definitions

The NPF_eventMask_t bits defined below is used for selecting the following IPv6 Unicast Forwarding Service API events

- NPF_IPV6UC_PREFIX_TBL_MISS
- NPF_IPV6UC_NEXT_HOP_TBL_MISS
- NPF_IPV6UC_FIB_PREFIX_MISS

- NPF_IPV6UC_FWDTBL_REFRESH
- NPF_IPV6UC_ADD_RES_TRANSITION

respectively.

```
/*
 * Definitions for selectively enabling IPV6UC events
 */
#define NPF_IPV6UC_EV_PREFIX_TBL_MISS_ENABLE    (1 << 0)
#define NPF_IPV6UC_EV_NEXT_HOP_TBL_MISS_ENABLE  (1 << 1)
#define NPF_IPV6UC_EV_FIB_PREFIX_MISS_ENABLE    (1 << 2)
#define NPF_IPV6UC_EV_FWDTBL_REFRESH_ENABLE     (1 << 3)
#define NPF_IPV6UC_EV_ADD_RES_TRANSITION_ENABLE (1 << 4)
#define NPF_IPV6UC_EV_LAST                      (1 << 4)
```

5 Function Calls

5.1 Completion Callback Function Calls

This callback function is for the application to register an asynchronous response handling routine to the IPv6Unicast API implementation. This callback function is intended to be implemented by the application, and to be registered to the IPv6 Unicast API implementation through the NPF_IPv6UC_Register function.

For more information regarding the design and usage of completion callbacks, please refer to Section 7, "Function Invocation Model, Events and Completion Callbacks", of the NPF Software Implementation Agreement - Software API Conventions (Revision 2, September 2003).

5.1.1 NPF_IPv6UC_CallbackFunc

Syntax

```
typedef void (*NPF_IPv6UC_CallbackFunc_t) (
   NPF_IN NPF_userContext_t         userContext,
   NPF_IN NPF_correlator_t          correlator,
   NPF_IN NPF_IPv6UC_CallbackData_t data);
```

Description

This function is a registered completion callback routine for handling IPv6 Unicast asynchronous responses.

This is a required function.

Input Arguments

- userContext - The context item that was supplied by the application when the completion callback routine was registered.
- correlator - The correlator item that was supplied by the application when the IPv6 Unicast API function call was invoked.
- data - The response information related to the particular IPv6 Unicast call, which is identified by the type field in the callback data.

Output Arguments

None

Return Values

None

5.2 Event Notification Function Calls

This event notification function is for the application to register an event handler routine to the IPv6Unicast API implementation. This handler function is intended to be implemented by the application, and to be registered to the IPv6 Unicast API implementation through the NPF_IPv6UC_EventRegister function.

5.2.1 NPF_IPv6UC_EventCallFunc_t

Syntax

```
typedef void (*NPF_IPv6UC_EventCallFunc_t) (
   NPF_IN NPF_userContext_t       userContext,
   NPF_IN NPF_IPv6UC_EventArray_t data);
```

Description

This function is a registered event notification routine for handling IPv6 Unicast events.

This is a required function.

Input Arguments

userContext - The context item that was supplied by the application when the event callback routine was registered.

data – A structure containing an array of event data structures and a count to indicate how many events are present. Each of these NPF_IPv6UC_EventData_t members contains event specific information and a type field to identify the particular event.

Output Arguments

None

Return Values

None

5.3 Callback Registration/Deregistration Function Calls

This section defines the registration and de-registration functions used to install and remove an asynchronous response callback routine.

5.3.1 NPF_IPv6UC_Register

Syntax

```
NPF_error_t NPF_IPv6UC_Register(
   NPF_IN NPF_userContext_t         userContext,
   NPF_IN NPF_IPv6UC_CallbackFunc_t callbackFunc,
   NPF_OUT NPF_callbackHandle_t    *callbackHandle);
```

Description

This function is used by an application to register its completion callback function for receiving asynchronous responses related to IPv6Unicast API function calls. Applications MAY register multiple callback functions using this function. The callback function is identified by the pair of userContext and callbackFunc, and for each individual pair, a unique callbackHandle will be assigned for future reference.

Since the callback function is identified by both userContext and callbackFunc, duplicate registration of the same callback function with a different userContext is allowed. Also, the same userContext can be shared among different callback functions. Duplicate registration of the same userContext and callbackFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_ALREADY_REGISTERED.

This is a required function.

Input Arguments

- userContext – A context item for uniquely identifying the context of the application registering the completion callback function. The exact value will be provided back to the registered completion callback function as its first parameter when it is called. Applications can assign any value to the userContext and the value is completely opaque to the IPv6Unicast API implementation.
- callbackFunc – The pointer to the completion callback function to be registered.

Output Arguments

- callbackHandle - A unique identifier assigned for the registered userContext and callbackFunc pair. This handle will be used by the application to specify which callback function to be called when invoking asynchronous NPF IPv6Unicast API functions. It will also be used when deregistering the userContext and callbackFunc pair.

Return Values

NPF_NO_ERROR - The registration completed successfully.

NPF_E_BAD_CALLBACK_FUNCTION – The callbackFunc is NULL, or otherwise invalid.

NPF_E_ALREADY_REGISTERED – No new registration was made since the userContext and callbackFunc pair was already registered.

Notes

- This API function MUST be invoked by any application interested in receiving asynchronous responses for IPv6 Unicast API function calls.
- This function operates in a synchronous manner, providing a return value as listed above.

5.3.2 NPF_IPv6UC_Deregister

Syntax

```
NPF_error_t NPF_IPv6UC_Deregister(
   NPF_IN NPF_callbackHandle_t callbackHandle);
```

Description

This function is used by an application to de-register a completion callback function, which was previously registered to handle asynchronous callbacks related to API function invocations.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier returned to the application when the completion callback routine was registered. It represents a unique user context and callback function pair.

Output Arguments

None

Return Values

NPF_NO_ERROR - The de-registration completed successfully.

NPF_E_BAD_CALLBACK_HANDLE – The de-registration did not complete successfully due to problems with the callback handle provided.

Notes

- This API function may be invoked by any application no longer interested in receiving asynchronous responses for IPv6 Unicast API function calls.
- This function operates in a synchronous manner, providing a return value as listed above.
- There may be a timing window where underlying modules may continue deliver outstanding callbacks to the callback routine after the de-registration function has been invoked. It is the API implementation's responsibility to guarantee that the callback function is not called after the deregister function has completed.

5.4 Event Registration/Deregistration Function Calls

This section defines the registration and de-registration functions used to install and remove an event handler routine

5.4.1 NPF_IPv6UC_EventRegister

Syntax

```
NPF_error_t NPF_IPv6UC_EventRegister(
   NPF_IN NPF_userContext_t          userContext,
   NPF_IN NPF_IPv6UC_EventCallFunc_t eventCallFunc,
   NPF_IN NPF_eventMask_t            eventMask,
   NPF_OUT NPF_callbackHandle_t     *eventCallHandle);
```

Description

This function is used by an application to register its event handling routine for receiving notifications of IPv6Unicast events. Applications MAY register multiple event handling routines using this function. The event handling routine is identified by the pair of userContext and eventCallFunc, and for each individual pair, a unique eventCallHandle will be assigned for future reference.

Since the event handling routine is identified by both userContext and eventCallFunc, duplicate registration of the same event handling routine with a different userContext is allowed. Also, the same userContext can be shared among different event handling routines. Duplicate registration of the same userContext and eventCallFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_ALREADY_REGISTERED.

This function also enables notifications for the events selected by the bits that are set in the eventMask parameter. A mask with all bits set (NPF_EV_ALL_EVENTS_ENABLE) selects all events of this SAPI. If the application wishes to change the selection of events, it may call the event registration function again with the same userContext and eventCallFunc, but with a different event selection mask. The events enabled are those whose bits were set in the most recent registration function call for a particular userContext and eventCallFunc pair.

This is a required function.

Input Arguments

- userContext – A context item for uniquely identifying the context of the application registering the event handling routine. The exact value will be provided back to the registered event handling routine as its first parameter when it is called. Applications can assign any value to the userContext and the value is completely opaque to the IPv6Unicast API implementation.
- eventCallFunc – The pointer to the event handling routine to be registered.
- eventMask – A bit-mask used to indicate which events the application wishes to receive.

Output Arguments

- eventCallHandle - A unique identifier assigned for the registered userContext and eventCallFunc pair. This handle will be used when deregistering the userContext and eventCallFunc pair.

Return Values

NPF_NO_ERROR - The registration completed successfully.

NPF_E_BAD_CALLBACK_FUNCTION – The eventCallFunc is NULL, or otherwise invalid.

NPF_E_CALLBACK_ALREADY_REGISTERED – No new registration was made since the userContext and eventCallFunc pair was already registered.

Notes

- This API function may be invoked by any application interested in receiving IPv6 Unicast events.
- This function operates in a synchronous manner, providing a return value as listed above.
- Even if a system implementation does not support events, the API implementation needs to implement this function to enable interoperability.

5.4.2 NPF_IPv6UC_EventDeregister

Syntax

```
NPF_error_t NPF_IPv6UC_EventDeregister(
   NPF_IN NPF_callbackHandle_t eventCallHandle);
```

Description

This function is used by an application to de-register an event handler routine which was previously registered to receive notifications of IPv6 Unicast events. It represents a unique user context and event handling routine pair.

This is a required function.

Input Arguments

- eventCallHandle - The unique identifier returned to the application when the event callback routine was registered.

Output Arguments

None

Return Values

NPF_NO_ERROR - The de-registration completed successfully.

NPF_E_BAD_CALLBACK_HANDLE – The de-registration did not complete successfully due to problems with the callback handle provided.

Notes

- This API function may be invoked by any application no longer interested in receiving IPv6 Unicast events.
- This function operates in a synchronous manner, providing a return value as listed above.
- There may be a timing window where outstanding events continue to be delivered to the event routine after the de-registration function has been invoked. It is the implementation's responsibility to guarantee that the event handler function is not called after the deregister function has returned.
- Even if an implementation does not support events, the implementation needs to implement this function to enable interoperability.

5.5 Supported & Preferred Mode Query Function Calls

These function calls are used by applications to query an implementation about what table modes are supported and which are preferred for best performance.

5.5.1 NPF_IPv6UC_GetSupportedModes

Syntax

```
NPF_IPv6UC_SupportedMode_t NPF_IPv6UC_GetSupportedModes();
```

Description

This function queries the supported table modes of an implementation.

This is a required function.

Input Argument

None

Output Arguments

None

Return Values

NPF_IPV6UC_UNIFIED_ONLY – The table implementation only supports a unified table mode, and will return NPF_E_FUNCTION_NOT_SUPPORTED if the prefix and next hop table manipulation functions are used.

NPF_IPV6UC_BOTH_SUPPORTED – The table implementation supports both a unified table mode and a discrete table mode.

Notes

None

Asynchronous Response

None

5.5.2 NPF_IPv6UC_GetPreferredMode

Syntax

```
NPF_IPv6UC_PreferredMode_t NPF_IPv6UC_GetPreferredMode();
```

Description

This function queries the preferred table modes of an implementation. If the supported mode call indicates that only a unified mode is supported, then this function call will return NPF_IPV6UC_UNIFIED_PREFERRED. However, if the supported mode call indicates that both modes are supported, then this function call may return any one of the three return values listed below.

This is a required function.

Input Argument

None

Output Arguments

None

Return Values

NPF_IPV6UC_DISCRETE_PREFERRED – The table implementation provides better performance when used with the discrete table APIs.

NPF_IPV6UC_UNIFIED_PREFERRED – The table implementation provides better performance when used with the unified table APIs.

NPF_IPV6UC_NO_PREFERENCE – The table implementation provides equally good or conditional performance when used with either API. Note that this value may only be returned if the supported mode call indicated both mode types are supported.

Notes

None

Asynchronous Response

None

5.6 *Unified FIB Table Function Calls*

This section specifies the functions defined to operate upon the unified mode FIB table.

5.6.1 NPF_IPv6UC_FibTableHandleCreate

Syntax

```
NPF_error_t NPF_IPv6UC_FibTableHandleCreate(
   NPF_IN NPF_callbackHandle_t    callbackHandle,
   NPF_IN NPF_correlator_t        correlator,
   NPF_IN NPF_errorReporting_t    errorReporting,
   NPF_IN NPF_IPv6UC_FibTableId_t fibTableId);
```

Description

This function creates internal and external handles for a FIB Table. The internal handle is used when calling IPv6 Unicast Forwarding API functions. The external handle is used when calling other functions in other APIs that need to refer to a forwarding table, regardless of whether it is managed in unified or discrete mode.

This is a required function.

Input Argument

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- fibTableId - A FIB Table id generated by the application. Must be nonzero and different from FIB Table id values of existing FIB Tables created on this API.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle creation did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_RESOURCE_EXISTS - A FIB Table with the same application assigned FIB Table id value already exists; no new FIB Table is created.

Notes

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

An **`NPF_IPv6UC_FibCreateResp_t`** structure, containing both internal and external handles, will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_EXISTS - A FIB Table with the same application assigned FIB Table id value already exists; its handles are returned in the callback, and no new FIB Table is created.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

5.6.2 NPF_IPv6UC_FibTableHandleDelete

Syntax

```
NPF_error_t NPF_IPv6UC_FibTableHandleDelete(
   NPF_IN NPF_callbackHandle_t          callbackHandle,
   NPF_IN NPF_correlator_t              correlator,
   NPF_IN NPF_errorReporting_t          errorReporting,
   NPF_IN NPF_IPv6UC_FibTableHandle_t    tableHandle);
```

Description

This function deletes a handle for a FIB Table. Subsequent use of the deleted handle in an API function call will result in an NPF_IPV6UC_E_INVALID_HANDLE error.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - The FIB table handle to delete.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle deletion did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the FIB table was detected. The FIB table was previously destroyed or never existed. No FIB table was deleted.

Notes

None

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the FIB table was detected. The FIB table was previously destroyed or never existed. No FIB table was deleted.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.6.3 NPF_IPv6UC_FibEntryAdd

Syntax

```
NPF_error_t NPF_IPv6UC_FibEntryAdd(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_FibTableHandle_t    tableHandle,
   NPF_IN NPF_uint32_t                   numEntries,
   NPF_IN NPF_IPv6UC_Prefix_t           *prefixArray,
   NPF_IN NPF_IPv6UC_NextHopArray_t     *nextHopArrays);
```

Description

This function may be used to insert one or more entries into a FIB table. The prefixArray and nextHopArrays fields point to arrays of size numEntries, where each element is positionally related.

If no table entry exists for each destination IPv6 address and length indicated in the prefixArray, then the prefix and next hop array information is added to create a new entry in the specified table.

If a table entry already exists, then the next hop array is replaced with the information specified in the associated element of the nextHopArrays array.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - FIB table identifier.
- numEntries – The number of elements in the prefixArray and the nextHopArrays. Each of these arrays has the same number of elements and they are positionally related.
- prefixArray – Pointer to the array of prefixes to add.
- nextHopArrays – Pointer to an array of NPF_IPv6UC_NextHopArray_t structures, which are associated with the prefixes. Each NPF_IPv6UC_NextHopArray_t structure contains a count plus one or more next hop.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not added to the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not added to the table because the callback handle was invalid.

Notes

When determining whether an entry is already present in the FIB, only the IPv6 address and prefix length are considered.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_Prefix_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and prefix length.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.6.4 NPF_IPv6UC_FibEntryDelete

Syntax

```
NPF_error_t NPF_IPv6UC_FibEntryDelete(
   NPF_IN NPF_callbackHandle_t            callbackHandle,
   NPF_IN NPF_correlator_t                correlator,
   NPF_IN NPF_errorReporting_t            errorReporting,
   NPF_IN NPF_IPv6UC_FibTableHandle_t     tableHandle,
   NPF_IN NPF_uint32_t                    numEntries,
   NPF_IN NPF_IPv6UC_Prefix_t            *prefixArray);
```

Description

All entries in the designated FIB table that match those found in the prefixArray will be removed.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - FIB table identifier.
- numEntries – The number of elements in the prefixArray.
- prefixArray – A pointer to an array of prefixes, one for each FIB table entry to be deleted.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not deleted from the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not deleted from the table because the callback handle was invalid.

Notes

When determining whether an entry is already present in the FIB table, only the IPv6 address and prefix length are considered.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_Prefix_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the corresponding IP address and prefix length.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.6.5 NPF_IPv6UC_FibTableFlush

Syntax

```
NPF_error_t NPF_IPv6UC_FibTableFlush(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_FibTableHandle_t    tableHandle);
```

Description

All entries in the designated FIB table will be removed and the designated FIB will be left empty.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - FIB table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not deleted from the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not deleted from the table because the callback handle was invalid.

Notes

This operation removes all entries from the specified FIB table, but does not destroy that FIB table.

If a FIB entry is removed, a reference to the removed entry by the forwarding plane MAY generate an NPF_IPV6UC_FIB_PREFIX_MISS event.

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.6.6 NPF_IPv6UC_FibTableAttributeQuery

Syntax

```
NPF_error_t NPF_IPv6UC_FibTableAttributeQuery(
   NPF_IN NPF_callbackHandle_t         callbackHandle,
   NPF_IN NPF_correlator_t             correlator,
   NPF_IN NPF_errorReporting_t         errorReporting,
   NPF_IN NPF_IPv6UC_FibTableHandle_t  tableHandle);
```

Description

This call will provide information about the characteristics of the specified FIB table. Currently, the attributes available are:

- An estimate of how many free entries are in this table.

This is an optional function. Implementations that do not support attribute queries MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation. The only valid error reporting for this method is NPF_REPORT_ALL.
- tableHandle – FIB table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The table was not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the FIB table. For example, prior to inserting any entries into the FIB table, an RTM might query the available free space of the FIB table and, therefore, be able to know when it cannot add any more entries to the table.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many “Add” requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the FIB table. The tableSpaceRemaining field in the NPF_IPv6UC_AsyncResponse_t struct will be set. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.6.7 NPF_IPv6UC_FibEntryQuery

Syntax

```
NPF_error_t NPF_IPv6UC_FibEntryQuery(
            NPF_IN NPF_callbackHandle_t          callbackHandle,
            NPF_IN NPF_correlator_t              correlator,
            NPF_IN NPF_errorReporting_t          errorReporting,
            NPF_IN NPF_IPv6UC_FibTableHandle_t   tableHandle,
            NPF_IN NPF_uint32_t                  numEntries,
            NPF_IN NPF_IPv6UC_Prefix_t          *prefixArray);
```

Description

This function call is used to query one or more FIB entries in the FIB table. If the entries exist, the content of the entries are returned in the completion callback.

This is an optional function. Implementations that do not support entry queries MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle - FIB table identifier.
- numEntries – The number of elements in the prefixArray.
- prefixArray – A pointer to an array of prefixes to query. Only the IP address and prefix length are considered in the key.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The query capability is not supported by this implementation.

Notes

None

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_FibQueryResp_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and prefix length.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. Because this function call will always return information that was requested, if all of the elements in the request array completed successfully and there is no additional data to return, the callback will return an allOK value of NPF_FALSE, a numResp value equal to the number of responses, and the array pointer pointing to the responses.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation’s choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.7 Discrete Prefix Table Function Calls

This section specifies the functions defined to operate upon the discrete mode prefix table.

5.7.1 NPF_IPv6UC_PrefixTableHandleCreate

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixTableHandleCreate(
   NPF_IN NPF_callbackHandle_t        callbackHandle,
   NPF_IN NPF_correlator_t            correlator,
   NPF_IN NPF_errorReporting_t        errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableId_t  prefixTableId);
```

Description

This function creates internal and external handles for a Prefix Table. The internal handle is used when calling IPv6 Unicast Forwarding API functions. The external handle is used when calling other functions in other APIs that need to refer to a forwarding table, regardless of whether it is managed in unified or discrete mode.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Argument

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- prefixTableId - A Prefix Table id generated by the application. Must be nonzero and different from Prefix Table id of existing Prefix Tables created on this API.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle creation did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

NPF_E_RESOURCE_EXISTS - A Prefix Table with the same application assigned Prefix Table id value already exists; no new Prefix Table is created.

Notes

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

An **`NPF_IPv6UC_PfxCreateResp_t`** structure, containing both internal and external handles, will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR – The operation completed successfully.

NPF_E_RESOURCE_EXISTS - A Prefix Table with the same application assigned Prefix Table id value already exists; its handles are returned in the callback, and no new Prefix Table is created.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE – The operation failed due to lack of resources.

5.7.2 NPF_IPv6UC_PrefixTableHandleDelete

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixTableHandleDelete(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableHandle_t tableHandle);
```

Description

This function deletes a handle for a Prefix Table. Subsequent use of the deleted handle in an API function call will result in an NPF_IPV6UC_E_INVALID_HANDLE error.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - The prefix table handle to delete.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle deletion did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the prefix table was detected. The prefix table was previously destroyed or never existed. No prefix table was deleted.

Notes

None

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the prefix table was detected. The prefix table was previously destroyed or never existed. No prefix table was deleted.

NPF_IPV6UC_E_INVALID_HANDLE – The operation did not complete successfully due to problems with the table handle.

5.7.3 NPF_IPv6UC_PrefixEntryAdd

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixEntryAdd(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableHandle_t tableHandle,
   NPF_IN NPF_uint32_t                   numEntries,
   NPF_IN NPF_IPv6UC_Prefix_t           *prefixArray,
   NPF_IN NPF_uint32_t                  *nextHopIdArray);
```

Description

This function may be used to insert one or more entries into a prefix table. The prefixArray and nextHopIdArray fields point to arrays of size numEntries, where each element is positionally related.

If no table entry exists for each destination IPv6 address and length indicated in the prefixArray, then the prefix and next hop identifier information is added to create a new entry in the specified table.

If a table entry already exists, then the next hop identifier is replaced with the information specified in the associated element of the nextHopIdArray.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle - Prefix table identifier.
- numEntries – The number of elements in the prefixArray and the nextHopIdArray. Each of these arrays has the same number of elements and they are positionally related.
- prefixArray - Pointer to the array of prefixes to add.
- nextHopIdArray – Pointer to the array of next hop identifiers associated with the prefixes.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not added to the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not added to the table because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Notes
When determining whether an entry is already present in the prefix table, only the IPv6 address and prefix length are considered.

Asynchronous Response
There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_Prefix_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and prefix length.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.7.4 NPF_IPv6UC_PrefixEntryDelete

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixEntryDelete(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableHandle_t tableHandle,
   NPF_IN NPF_uint32_t                   numEntries,
   NPF_IN NPF_IPv6UC_Prefix_t           *prefixArray);
```

Description

This function may be used to remove one or more entries from a prefix table. If a prefix table entry exists as indicated by the destination IPv6 address and prefix length in an element contained in the prefixArray, then that entry will be removed from the specified table.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle - Prefix table identifier.
- numEntries – The number of elements in the prefixArray.
- prefixArray - A pointer to an array of prefixes, one for each prefix table entry to be deleted.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not deleted from the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not deleted from the table because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Notes

When determining whether an entry is already present in the prefix table, only the IPv6 address and prefix length are considered.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_Prefix_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST – The operation did not complete successfully since the specified entry was not found.

The response array returned in the call back may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and prefix length.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function calls.

5.7.5 NPF_IPv6UC_PrefixTableFlush

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixTableFlush(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableHandle_t tableHandle);
```

Description

All entries in the designated prefix table will be removed and the designated table will be left empty.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle - Prefix table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The prefix table was not flushed due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The prefix table was not flushed because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Notes

This operation removes all entries from the specified table, but does not destroy that table.

If a prefix table entry is removed, a reference to the removed entry by the forwarding plane MAY generate an NPF_IPV6UC_PREFIX_TBL_MISS event.

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.7.6 NPF_IPv6UC_PrefixTableAttributeQuery

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixTableAttributeQuery(
   NPF_IN NPF_callbackHandle_t          callbackHandle,
   NPF_IN NPF_correlator_t              correlator,
   NPF_IN NPF_errorReporting_t          errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableHandle_t tableHandle);
```

Description

This call will provide information about the characteristics of the specified prefix table. Currently, the attributes available are:

- An estimate of how many free entries are in this table.

This is an optional function. Implementations that do not support queries or that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation. The only valid error reporting for this call is NPF_REPORT_ALL.
- tableHandle - Prefix table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The table was not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The attribute query capability or discrete table operations are not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the prefix table. For example, prior to inserting any prefix entries into the prefix table, an RTM might query the available free space of the prefix table and, therefore, be able to know when it cannot add any more entries to the table.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many “Add” requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response
A return code will be returned asynchronously along with an approximation of the number of free entries left in the prefix table. The tableSpaceRemaining field in the NPF_IPv6UC_AsyncResponse_t struct will be set. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.7.7 NPF_IPv6UC_PrefixEntryQuery

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixEntryQuery(
        NPF_IN NPF_callbackHandle_t          callbackHandle,
        NPF_IN NPF_correlator_t              correlator,
        NPF_IN NPF_errorReporting_t          errorReporting,
        NPF_IN NPF_IPv6UCPrefixTableHandle_t tableHandle,
        NPF_IN NPF_uint32_t                  numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t          *prefixArray);
```

Description

This function call is used to query one or more prefix entries in the prefix table. If the entries exist, the content of the entries are returned in the completion callback.

This is an optional function. Implementations that do not support queries or that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API invocation.
- tableHandle - Prefix table identifier.
- numEntries – The number of elements in the prefixArray.
- prefixArray - Pointer to the array of prefixes to query. Only the address and prefix length are considered in the key.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The entry query capability or discrete table operations are not supported by this implementation.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_PrefixQueryResp_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the call back may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and prefix length.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. Because this function call will always return information that was requested, if all of the elements in the request array completed successfully and there is no additional data to return, the callback will return an allOK value of NPF_FALSE, a numResp value equal to the number of responses, and the array pointer pointing to the responses.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function calls.

5.7.8 NPF_IPv6UC_PrefixNextHopTableBind

Syntax

```
NPF_error_t NPF_IPv6UC_PrefixNextHopTableBind(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_PrefixTableHandle_t  prefixTableHandle),
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t nextHopTableHandle);
```

Description

This function makes an association between a Prefix Table and a Next Hop Table. It designates the Next Hop Table whose entries are to be used when a particular Prefix Table is referenced. If the Prefix Table is already associated with another Next Hop Table, that association is replaced by the new Next Hop Table. If the Next Hop Table is already associated with a different Prefix Table, the new Prefix Table is added to the set of Prefix Tables that share this Next Hop Table. Thus the possible relationships of Prefix Table to Next Hop Table are one-to-one and many-to-one, but never one-to-many.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- prefixTableHandle - The prefix table identifier.
- nextHopTableHandle - The next hop table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table binding did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Notes

None

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE – The operation did not complete successfully due to problems with one of the table handles.

5.8 Discrete Next Hop Table Function Calls

This section specifies the functions defined to operate upon the discrete mode next hop table.

5.8.1 NPF_IPv6UC_NextHopTableHandleCreate

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopTableHandleCreate(
   NPF_IN NPF_callbackHandle_t        callbackHandle,
   NPF_IN NPF_correlator_t            correlator,
   NPF_IN NPF_errorReporting_t        errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableId_t nextHopTableId);
```

Description

This function creates a handle for a Next Hop Table.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Argument

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- nextHopTableId - A Next Hop Table id generated by the application. Must be nonzero and different from Next Hop Table id values of existing Next Hop Tables created on this API.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle creation did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

NPF_E_RESOURCE_EXISTS - A Next Hop Table with the same application assigned Next Hop Table id value already exists; no new Next Hop Table is created.

Notes

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

A next hop table handle will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_EXISTS - A Next Hop Table with the same application assigned Next Hop Table id value already exists; its handle is returned in the callback, and no new Next Hop Table is created.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

5.8.2 NPF_IPv6UC_NextHopTableHandleDelete

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopTableHandleDelete(
   NPF_IN NPF_callbackHandle_t            callbackHandle,
   NPF_IN NPF_correlator_t                correlator,
   NPF_IN NPF_errorReporting_t            errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t tableHandle);
```

Description

This function deletes a handle for a Next Hop Table. Subsequent use of the deleted handle in an API function call will result in an NPF_IPV6UC_E_INVALID_HANDLE error.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - The next hop table handle to delete.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle deletion did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the next hop table was detected. The next hop table was previously destroyed or never existed. No next hop table was deleted.

Notes

None

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the next hop table was detected. The next hop table was previously destroyed or never existed. No next hop table was deleted.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.8.3 NPF_IPv6UC_NextHopEntryAdd

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopEntryAdd(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t tableHandle,
   NPF_IN NPF_uint32_t                   numEntries,
   NPF_IN NPF_uint32_t                  *nextHopIdArray,
   NPF_IN NPF_IPv6UC_NextHopArray_t      *nextHopArrays);
```

Description

This function may be used to insert one or more entries into a next hop table. The nextHopIdArray and nextHopArrays fields point to arrays of size numEntries, where each element is positionally related.

If no table entry exists for each next hop identifier indicated in the nextHopIdArray, then the next hop identifier and next hop array information is added to create a new entry in the specified table.

If a table entry already exists, then the next hop array is replaced with the information specified in the associated element of the nextHopArrays array.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle – Next Hop Table identifier.
- numEntries – The number of elements in the nextHopIdArray and the nextHopArrays. Each of these arrays has the same number of elements and they are positionally related.
- nextHopIdArray – Pointer to an array of next hop identifiers.
- nextHopArrays - Pointer to an array of NPF_IPv6UC_NextHopArray_t structures, which are associated with the next hop identifiers. Each NPF_IPv6UC_NextHopArray_t structure contains a count plus one or more next hop definitions.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not added to the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not added to the table because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Notes
None

Asynchronous Response
There may be multiple asynchronous callbacks to this request. The Next Hop Identifier will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the Next Hop Identifier value.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.8.4 NPF_IPv6UC_NextHopEntryDelete

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopEntryDelete(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t tableHandle,
   NPF_IN NPF_uint32_t                   numEntries,
   NPF_IN NPF_uint32_t                  *nextHopIdArray);
```

Description

This function deletes one or more Next Hop Entries. If a Next Hop Entry exists, that entry will be removed from the specified table.

If a Next Hop Entry is removed, a reference to the removed entry by the forwarding plane MAY generate an NPF_IPV6UC_NEXT_HOP_TBL_MISS event.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle – Next Hop Table identifier.
- numEntries – The number of elements in the nextHopIdArray.
- nextHopIdArray – A pointer to an array of Next Hop Identifier values, one for each next hop table entry to be deleted.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not deleted from the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not deleted from the table because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. The Next Hop Identifier will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the next hop identifier value.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.8.5 NPF_IPv6UC_NextHopTableFlush

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopTableFlush(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t tableHandle);
```

Description

All entries in the designated next hop table will be removed and the designated table will be left empty.

This is an optional function. Implementations that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle – Next Hop Table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table was not flushed due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The table was not flushed because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – Discrete table operations are not supported by this implementation.

Notes

This operation removes all entries from the specified table, but does not destroy that table.

If a Next Hop Entry is removed, a reference to the removed entry by the forwarding plane MAY generate an NPF_IPV6UC_NEXT_HOP_TBL_MISS event.

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.8.6 NPF_IPv6UC_NextHopTableAttributeQuery

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopTableAttributeQuery(
   NPF_IN NPF_callbackHandle_t            callbackHandle,
   NPF_IN NPF_correlator_t                correlator,
   NPF_IN NPF_errorReporting_t            errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t tableHandle);
```

Description

This call will provide information about the characteristics of the specified Next Hop Table. Currently, the attributes available are:

- An estimate of how many free entries are in this table.

This is an optional function. Implementations that do not support queries or that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation. The only valid reporting level is NPF_REPORT_ALL.
- tableHandle – The Next Hop Table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The table was not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The attribute query capability or discrete table operations are not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the Next Hop Table. For example, prior to inserting any next hop entries into the next hop table, an RTM might query the available free space of the Next Hop Table and, therefore, be able to know when it cannot add any more entries to the table.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many “Add” requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the Next Hop Table. The tableSpaceRemaining field in the NPF_IPv6UC_AsyncResponse_t struct will be set. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.8.7 NPF_IPv6UC_NextHopEntryQuery

Syntax

```
NPF_error_t NPF_IPv6UC_NextHopEntryQuery(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_NextHopTableHandle_t tableHandle,
   NPF_IN NPF_uint32_t                   numEntries,
   NPF_IN NPF_uint32_t                  *nextHopIdArray);
```

Description

This function call is used to query one or more next hop entries in the next hop table. If the entries exist, the content of the entries are returned in the completion callback.

This is an optional function. Implementations that do not support queries or that do not support a discrete table mode MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API invocation.
- tableHandle – Next hop table identifier.
- numEntries – The number of elements in the nextHopIdArray.
- nextHopIdArray - Pointer to the array of next hop identifiers to query. Only the next hop identifier is considered in the key.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The query capability or discrete table operations are not supported by this implementation.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_NextHopQueryResp_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the next hop identifier value.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. Because this function call will always return information that was requested, if all of the elements in the request array completed successfully and there is no additional data to return, the callback will return an allOK value of NPF_FALSE, a numResp value equal to the number of responses, and the array pointer pointing to the responses.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.9 Address Resolution Function Calls

This section specifies the functions defined to operate upon the address resolution table. These functions are intended to be used in either unified or discrete modes.

5.9.1 NPF_IPv6UC_AddResTableHandleCreate

Syntax

```
NPF_error_t NPF_IPv6UC_AddResTableHandleCreate(
   NPF_IN NPF_callbackHandle_t        callbackHandle,
   NPF_IN NPF_correlator_t            correlator,
   NPF_IN NPF_errorReporting_t        errorReporting,
   NPF_IN NPF_IPv6UC_AddResTableId_t  addResTableId);
```

Description

This function creates a handle for an Address Resolution Table.

This is a required function.

Input Argument

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- AddResTableId - An Address Resolution Table id generated by the application. Must be nonzero and different from Address Resolution Table identifiers of existing Address Resolution Tables created on this API.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle creation did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_RESOURCE_EXISTS - An Address Resolution Table with the same application assigned Address Resolution Table id value already exists; no new Address Resolution Table is created.

Notes

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

An address resolution table handle will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_EXISTS - An Address Resolution Table with the same application assigned Address Resolution Table id value already exists; its handle is returned in the callback, and no new Address Resolution Table is created.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

5.9.2 NPF_IPv6UC_AddResTableHandleDelete

Syntax

```
NPF_error_t NPF_IPv6UC_AddResTableHandleDelete(
   NPF_IN NPF_callbackHandle_t           callbackHandle,
   NPF_IN NPF_correlator_t               correlator,
   NPF_IN NPF_errorReporting_t           errorReporting,
   NPF_IN NPF_IPv6UC_AddResTableHandle_t tableHandle);
```

Description

This function deletes a handle for an Address Resolution Table. Subsequent use of the deleted handle in an API function call will result in an NPF_IPV6UC_E_INVALID_HANDLE error.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle - The address resolution table handle to delete.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table handle deletion did not complete successfully due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the address resolution table was detected. The address resolution table was previously destroyed or never existed. No address resolution table was deleted.

Notes

None

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_E_RESOURCE_NONEXISTENT - A duplicate request to destroy or free the address resolution table was detected. The address resolution table was previously destroyed or never existed. No address resolution table was deleted.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.9.3 NPF_IPv6UC_AddResEntryAdd

Syntax

```
NPF_error_t NPF_IPv6UC_AddResEntryAdd(
            NPF_IN NPF_callbackHandle_t                callbackHandle,
            NPF_IN NPF_correlator_t                    correlator,
            NPF_IN NPF_errorReporting_t                errorReporting,
            NPF_IN NPF_IPv6UC_AddResTableHandle_t      tableHandle,
            NPF_IN NPF_uint32_t                        numEntries,
            NPF_IN NPF_IPv6UC_AddResEntry_t           *entryArray);
```

Description

This function may be used to insert one or more entries into an address resolution table. The entryArray field points to an array of size numEntries, where each element is an address resolution entry to add.

If no table entry exists for the IP address and interface pair supplied in the entryArray, then the address resolution entry information is added to create a new entry in the specified table.

If a table entry already exists, then it is replaced with the information specified in the entryArray.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle – Address Resolution table identifier.
- numEntries – The number of elements in the entryArray.
- entryArray - Pointer to an array of NPF_IPv6UC_AddResEntry_t structures. Each structure has an IP address, logical interface handle and a media specific address.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not added due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The entries were not added to the table because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_AddResKey_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_INSUFFICIENT_STORAGE - The operation failed due to lack of resources.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing IP_Address and interfaceHandle fields.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

5.9.4 NPF_IPv6UC_AddResEntryDelete

Syntax

```
NPF_error_t NPF_IPv6UC_AddResEntryDelete(
            NPF_IN NPF_callbackHandle_t           callbackHandle,
            NPF_IN NPF_correlator_t               correlator,
            NPF_IN NPF_errorReporting_t           errorReporting,
            NPF_IN NPF_IPv6UC_AddResTableHandle_t tableHandle,
            NPF_IN NPF_uint32_t                   numEntries,
            NPF_IN NPF_IPv6UC_AddResKey_t        *entryArray);
```

Description

If an entry exists in the address resolution table as indicated by the IP address and interface pair supplied in an Address Resolution entry contained in the entryArray, then it will be removed from the specified table.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- tableHandle – Address Resolution table handle.
- numEntries – The number of elements in the entryArray.
- entryArray – A pointer to an array of NPF_IPv6UC_AddResKey_t structures, one for each address resolution table entry to be deleted.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not deleted from the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE – The entries were not deleted from the table because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_AddResKey_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the call back may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and interface handle fields.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully and there is no additional response data to return, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function calls.

5.9.5 NPF_IPv6UC_AddResTableFlush

Syntax

```
NPF_error_t NPF_IPv6UC_AddResTableFlush(
   NPF_IN NPF_callbackHandle_t            callbackHandle,
   NPF_IN NPF_correlator_t                correlator,
   NPF_IN NPF_errorReporting_t            errorReporting,
   NPF_IN NPF_IPv6UC_AddResTableHandle_t   tableHandle);
```

Description

All entries in the designated address resolution table will be removed and the designated address resolution table will be left empty.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function call.
- tableHandle – Address resolution table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not deleted from the table due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not deleted from the table because the callback handle was invalid.

Notes

This operation removes all entries from the specified table, but does not destroy that table.

Asynchronous Response

A return code will be returned asynchronously. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.9.6 NPF_IPv6UC_AddResTableAttributeQuery

Syntax

```
NPF_error_t NPF_IPv6UC_AddResAttributeQuery(
   NPF_IN NPF_callbackHandle_t             callbackHandle,
   NPF_IN NPF_correlator_t                 correlator,
   NPF_IN NPF_errorReporting_t             errorReporting,
   NPF_IN NPF_IPv6UC_AddResTableHandle_t   tableHandle);
```

Description

This call will provide information about the characteristics of the specified address resolution table. Currently, the attributes available are:

- An estimate of how many free entries are in this table.

This is an optional function. Implementations that do not support queries MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation. The only valid error reporting for this method is NPF_REPORT_ALL.
- tableHandle – Address resolution table identifier.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The table was not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the address resolution table. For example, prior to inserting any address resolution entries into the table, the application might query the available free space of the address resolution table and, therefore, be able to know when it cannot add any more entries to the table.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many “Add” requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the address resolution table. The tableSpaceRemaining field in the NPF_IPv6UC_AsyncResponse_t struct will be set. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

5.9.7 NPF_IPv6UC_AddResEntryQuery

Syntax

```
NPF_error_t NPF_IPv6UC_AddResEntryQuery(
            NPF_IN NPF_callbackHandle_t           callbackHandle,
            NPF_IN NPF_correlator_t               correlator,
            NPF_IN NPF_errorReporting_t           errorReporting,
            NPF_IN NPF_IPv6UC_AddResTableHandle_t tableHandle,
            NPF_IN NPF_uint32_t                   numEntries,
            NPF_IN NPF_IPv6UC_AddResKey_t        *entryArray);
```

Description

This function call is used to query one or more address resolution entries in the address resolution table. If the entries exist, the content of the entries are returned in the completion callback.

This is an optional function. Implementations that do not support queries MUST implement a stub of this function and MUST immediately return NPF_E_FUNCTION_NOT_SUPPORTED when called.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API invocation.
- tableHandle – Address resolution table identifier.
- numEntries – The number of entries in the entryArray.
- entryArray - Pointer to the array of address resolution keys to query.

Output Arguments

None

Return Values

NPF_NO_ERROR - The operation is in progress.

NPF_E_UNKNOWN - The entries were not queried due to problems encountered when handling the input parameters.

NPF_E_BAD_CALLBACK_HANDLE - The entries were not queried because the callback handle was invalid.

NPF_E_FUNCTION_NOT_SUPPORTED – The query capability is not supported by this implementation.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. An **`NPF_IPv6UC_AddResQueryResp_t`** structure will be returned along with a return code. Possible return codes are:

NPF_NO_ERROR - The operation completed successfully.

NPF_IPV6UC_E_INVALID_HANDLE - The operation did not complete successfully due to problems with the table handle.

NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST - The operation did not complete successfully since the specified entry was not found.

The response array returned in the call back may contain between zero and the number of elements requested with this API function call. Each element in the response array can be correlated with an element in the request array by comparing the IP address and interface handle fields.

An NPF_IPv6UC_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_IPv6UC_AsyncResponse_t structures will also be returned. Because this function call will always return information that was requested, if all the elements in the request array completed successfully and there is no additional data to return, the callback will return an allOK value of NPF_FALSE, a numResp value equal to the number of responses, and the array pointer pointing to the responses.

If not all of the responses are complete or if not all of the responses were successful or if there is additional response data to return, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null. Failing elements may be determined by examining the return code in each array element.

It is the implementation’s choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function calls.

6 API Summary

These tables are included as a summary for informative purposes.

6.1 Common Function Calls

API function Name	Function Required
NPF_IPv6UC_Register	Required
NPF_IPv6UC_Deregister	Required
NPF_IPv6UC_EventRegister	Required
NPF_IPv6UC_EventDeregister	Required
NPF_IPv6UC_GetSupportedModes	Required
NPF_IPv6UC_GetPreferredMode	Required
NPF_IPv6UC_AddResTableHandleCreate	Required
NPF_IPv6UC_AddResTableHandleDelete	Required
NPF_IPv6UC_AddResEntryAdd	Required
NPF_IPv6UC_AddResEntryDelete	Required
NPF_IPv6UC_AddResTableFlush	Required
NPF_IPv6UC_AddResTableAttributeQuery	Optional
NPF_IPv6UC_AddResEntryQuery	Optional

6.2 Unified Mode Function Calls

API function Name	Function Required
NPF_IPv6UC_FibTableHandleCreate	Required
NPF_IPv6UC_FibTableHandleDelete	Required
NPF_IPv6UC_FibEntryAdd	Required
NPF_IPv6UC_FibEntryDelete	Required
NPF_IPv6UC_FibTableFlush	Required
NPF_IPv6UC_FibTableAttributeQuery	Optional
NPF_IPv6UC_FibEntryQuery	Optional

6.3 Discrete Mode Function Calls

If the Discrete Mode is implemented then all the functions below except NPF_IPv6UC_PrefixTableAttributeQuery, NPF_IPv6UC_PrefixEntryQuery, NPF_IPv6UC_NextHopTableAttributeQuery and NPF_IPv6UC_NextHopEntryQuery are Required.

API function Name	**Function Required**
`NPF_IPv6UC_PrefixTableHandleCreate`	Optional
`NPF_IPv6UC_PrefixTableHandleDelete`	Optional
`NPF_IPv6UC_PrefixEntryAdd`	Optional
`NPF_IPv6UC_PrefixEntryDelete`	Optional
`NPF_IPv6UC_PrefixTableFlush`	Optional
`NPF_IPv6UC_PrefixTableAttributeQuery`	Optional
`NPF_IPv6UC_PrefixNextHopTableBind`	Optional
`NPF_IPv6UC_PrefixEntryQuery`	Optional
`NPF_IPv6UC_NextHopTableHandleCreate`	Optional
`NPF_IPv6UC_NextHopTableHandleDelete`	Optional
`NPF_IPv6UC_NextHopEntryAdd`	Optional
`NPF_IPv6UC_NextHopEntryDelete`	Optional
`NPF_IPv6UC_NextHopTableFlush`	Optional
`NPF_IPv6UC_NextHopTableAttributeQuery`	Optional
`NPF_IPv6UC_NextHopEntryQuery`	Optional

6.4 Events

Event Name	**Event Required**
`NPF_IPV6UC_PREFIX_TBL_MISS`	Optional
`NPF_IPV6UC_NEXT_HOP_TBL_MISS`	Optional
`NPF_IPV6UC_FIB_PREFIX_MISS`	Optional
`NPF_IPV6UC_FWDTBL_REFRESH`	Optional
`NPF_IPV6UC_ADD_RES_TRANSITION`	Optional

7 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 Service API Revision 1.0

Appendix A   Header File: NPF_IPv6UC.h

```
/*
 * This header file defines typedefs, constants, and functions
 * that apply to the NPF IPv6 Unicast Forwarding Service API
 */
#ifndef __NPF_IPV6U_H
#define __NPF_IPV6U_H

#ifdef __cplusplus
extern "C"      {
#endif

/*----------------------------------------------------------------
 *
 * Common Data Types
 *
 *----------------------------------------------------------------*/

/*
 * Table support enumeration.
 */
typedef enum {
        NPF_IPV6UC_UNIFIED_ONLY         =  1,
        NPF_IPV6UC_BOTH_SUPPORTED       =  2
} NPF_IPv6UC_SupportedMode_t;

/*
 * Table preference enumeration. No preference value may only be returned
 * by implementations that returned "both supported" to the support query.
 */
typedef enum {
        NPF_IPV6UC_DISCRETE_PREFERRED   =  1,
        NPF_IPV6UC_UNIFIED_PREFERRED    =  2,
        NPF_IPV6UC_NO_PREFERENCE        =  3
} NPF_IPv6UC_PreferredMode_t;

/*
 * Prefix definition
 *
 * This structure is defined in a common NPF header file since it
 * is used by several APIs. It is replicated here as a comment for
 * informative purposes.
 */
/*
typedef struct {
        NPF_IPv6Address_t  IPv6Addr;
        NPF_uint8_t        IPv6Plen;
} NPF_IPv6Prefix_t;
*/

/*
 * Prefix retype definition
 */
typedef NPF_IPv6Prefix_t  NPF_IPv6UC_Prefix_t;
```

```
/*
 * Next hop type definition
 */
typedef enum {
        NPF_IPV6UC_NH_BASIC              =  1,
        NPF_IPV6UC_NH_DIRECT_ATTACH      =  2,
        NPF_IPV6UC_NH_SEND_TO_CP         =  3,
        NPF_IPV6UC_NH_DISCARD            =  4,
        NPF_IPV6UC_NH_REMOTE             =  5,
        NPF_IPV6UC_NH_TUNNEL             =  6
} NPF_IPv6UC_NextHopType_t;

/*
 * Media type definition:
 */
typedef  enum {
        NPF_NO_MEDIA_TYPE       =  1,
        NPF_MAC_ADDRESS         =  2,
        NPF_ATM_VC              =  3
} NPF_MediaType_t;

/*
 * Media Address structure:
 */
typedef struct {
        NPF_MediaType_t         type;
        union {
          NPF_MAC_Address_t     MAC_Address;
          NPF_VccAddr_t         ATM_Vc;
        }u;
}  NPF_MediaAddress_t;

/*
 * IPv6 unicast Address Resolution reachability type definition:
 */
typedef  enum {
      NPF_IPv6_VOID             =  0,
      NPF_IPv6_NONE             =  1,
      NPF_IPv6_INCOMPLETE       =  2,
      NPF_IPv6_REACHABLE        =  3,
      NPF_IPv6_STALE            =  4,
      NPF_IPv6_DELAY            =  5,
      NPF_IPv6_PROBE            =  6
} NPF_IPv6_Reachability_t;

/*
 * IPv6 unicast next hop structure: weight, egressInterface and
 * nextHopIP fields are valid only for IPV6UC_BASIC,
 * IPV6UC_DIRECT_ATTACH, IPV6UC_REMOTE, and IPV6_TUNNEL types.
 */
typedef struct {
        NPF_IPv6UC_NextHopType_t        type;
        NPF_uint16_t                    weight;
        NPF_IfHandle_t                  egressInterface;
        NPF_IPv6Address_t               nextHopIP;
        NPF_MediaAddress_t              mediaAddress;
        NPF_IPv6_Reachability_t         reachability;
} NPF_IPv6UC_NextHop_t;

/*
 * IPv6 unicast Next Hop Entry: nextHopArray points to an array
 * (one or more) of NPF_IPv6UC_NextHop_t structures.
 * nextHopCount indicates how many next hops are in the array.
 * An array is passed because a single prefix may use multiple
 * next hops.
 */
typedef struct {
        NPF_uint32_t             nextHopCount;
        NPF_IPv6UC_NextHop_t    *nextHopArray;
} NPF_IPv6UC_NextHopArray_t;

/*
 * IPv6 unicast Address Resolution entry:
 */
typedef struct {
        NPF_IPv6Address_t       IP_Address;
        NPF_IfHandle_t          interfaceHandle;
        NPF_MediaAddress_t      mediaAddress;
        NPF_IPv6_Reachability_t reachability;
} NPF_IPv6UC_AddResEntry_t;

/*
 * This structure contains the key of an Address Resolution
 * table entry, consisting of IP address and interface handle.
 */
typedef struct {
    NPF_IPv6Address_t           IP_Address;
    NPF_IfHandle_t              interfaceHandle;
} NPF_IPv6UC_AddResKey_t;

/*
 * Meaningful structure name used in adress resolution
 * asynchronous callback data.
 */

typedef NPF_IPv6UC_AddResEntry_t  NPF_IPv6UC_AddResQueryResp_t;

/*
 * Common table id types
 */
typedef NPF_uint32_t NPF_IPv6UC_AddResTableId_t;

/*
 * Common table handle types
 */
typedef NPF_uint32_t NPF_IPv6UC_AddResTableHandle_t;

/*
 * Forwarding Table Handle definition
 *
 * This structure is defined in a common NPF header file since it
 * is used by several APIs. It is replicated here as a comment for
 * informative purposes.
 */
```

```
/*
typedef NPF_uint32_t NPF_IPv6UC_FwdTableHandle_t;
*/


/*
 * Asynchronous error codes (returned in function callbacks)
 */

typedef NPF_uint32_t NPF_IPv6UC_ReturnCode_t;

#define IPV6_ERR(n) ((NPF_IPv6UC_ReturnCode_t) NPF_IPV6_BASE_ERR + (n))

#define NPF_IPV6UC_E_TABLE_ENTRY_DOES_NOT_EXIST           IPV6_ERR(1)
#define NPF_IPV6UC_E_INVALID_HANDLE                       IPV6_ERR(3)
#define NPF_IPV6UC_E_INSUFFICIENT_STORAGE                 IPV6_ERR(4)

/*----------------------------------------------------------------
 *
 * Discrete Mode Data Types
 *
 *----------------------------------------------------------------*/

/*
 * This structure contains the query results for a single
 * prefix table entry.
 */
typedef struct {
        NPF_IPv6UC_Prefix_t     prefix;
        NPF_uint32_t            nextHopIdentifier;
} NPF_IPv6UC_PrefixQueryResp_t;

/*
 * This structure contains the query results for a single
 * next hop table entry.
 */
typedef struct {
        NPF_uint32_t                    nextHopIdentifier;
        NPF_IPv6UC_NextHopArray_t       nextHopArray;
} NPF_IPv6UC_NextHopQueryResp_t;

/*
 * Discrete mode resource id types
 */
typedef NPF_uint32_t NPF_IPv6UC_PrefixTableId_t;
typedef NPF_uint32_t NPF_IPv6UC_NextHopTableId_t;

/*
 * Discrete mode handle types
 */
typedef NPF_uint32_t NPF_IPv6UC_PrefixTableHandle_t;
typedef NPF_uint32_t NPF_IPv6UC_NextHopTableHandle_t;

/*
 * Asynchronous response structure for NPF_IPv6UC_PrefixTableHandleCreate()
 */
typedef struct {
        NPF_IPv6UC_FwdTableHandle_t     extHandle;
        NPF_IPv6UC_PrefixTableHandle_t  intHandle;
} NPF_IPv6UC_PfxCreateResp_t;

/*----------------------------------------------------------------
 *
 * Unified Mode Data Types
 *
 *----------------------------------------------------------------*/

/*
 * This structure contains the query results for a single FIB table
 * entry.
 */
typedef struct {
        NPF_IPv6UC_Prefix_t             prefix;
        NPF_IPv6UC_NextHopArray_t       nextHopArray;
} NPF_IPv6UC_FibQueryResp_t;

/*
 * Unified table id types
 */
typedef NPF_uint32_t NPF_IPv6UC_FibTableId_t;

/*
 * Unified table handle types
 */
typedef NPF_uint32_t NPF_IPv6UC_FibTableHandle_t;

/*
 * Asynchronous response structure for NPF_IPv6UC_FIBTableHandleCreate()
 */
typedef struct {
        NPF_IPv6UC_FwdTableHandle_t     extHandle;
        NPF_IPv6UC_FibTableHandle_t     intHandle;
} NPF_IPv6UC_FibCreateResp_t;

/*----------------------------------------------------------------
 *
 * Completion Callback Data Types
 *
 *----------------------------------------------------------------*/

/*
 * Common callback definition:
 */
typedef enum NPF_IPv6UC_CallbackType {
        NPF_IPV6UC_PREFIX_TABLE_HANDLE_CREATE         = 1,
        NPF_IPV6UC_PREFIX_TABLE_HANDLE_DELETE         = 2,
        NPF_IPV6UC_PREFIX_ENTRY_ADD                   = 3,
        NPF_IPV6UC_PREFIX_ENTRY_DELETE                = 4,
        NPF_IPV6UC_PREFIX_TABLE_FLUSH                 = 5,
        NPF_IPV6UC_PREFIX_TABLE_ATTRIBUTE_QUERY       = 6,
        NPF_IPV6UC_PREFIX_ENTRY_QUERY                 = 7,
        NPF_IPV6UC_PREFIX_NEXT_HOP_TABLE_BIND         = 8,
        NPF_IPV6UC_NEXT_HOP_TABLE_HANDLE_CREATE       = 9,
        NPF_IPV6UC_NEXT_HOP_TABLE_HANDLE_DELETE       = 10,
        NPF_IPV6UC_NEXT_HOP_ENTRY_ADD               = 11,
        NPF_IPV6UC_NEXT_HOP_ENTRY_DELETE            = 12,
        NPF_IPV6UC_NEXT_HOP_TABLE_FLUSH             = 13,
        NPF_IPV6UC_NEXT_HOP_TABLE_ATTRIBUTE_QUERY   = 14,
        NPF_IPV6UC_NEXT_HOP_ENTRY_QUERY             = 15,
        NPF_IPV6UC_FIB_TABLE_HANDLE_CREATE          = 16,
        NPF_IPV6UC_FIB_TABLE_HANDLE_DELETE          = 17,
        NPF_IPV6UC_FIB_ENTRY_ADD                    = 18,
        NPF_IPV6UC_FIB_ENTRY_DELETE                 = 19,
        NPF_IPV6UC_FIB_TABLE_FLUSH                  = 20,
        NPF_IPV6UC_FIB_TABLE_ATTRIBUTE_QUERY        = 21,
        NPF_IPV6UC_FIB_ENTRY_QUERY                  = 22,
        NPF_IPV6UC_ADDRESS_RES_TABLE_HANDLE_CREATE  = 23,
        NPF_IPV6UC_ADDRESS_RES_TABLE_HANDLE_DELETE  = 24,
        NPF_IPV6UC_ADDRESS_RES_ENTRY_ADD            = 25,
        NPF_IPV6UC_ADDRESS_RES_ENTRY_DELETE         = 26,
        NPF_IPV6UC_ADDRESS_RES_TABLE_FLUSH          = 27,
        NPF_IPv6UC_ADDRESS_RES_TABLE_ATTRIBUTE_QUERY = 28,
        NPF_IPV6UC_ADDRESS_RES_ENTRY_QUERY          = 29
} NPF_IPv6UC_CallbackType_t;

/*
 * An asynchronous response contains a return code indicating
 * an error or success of a particular request operation.
 * The structure may also contain other optional information
 * that was requested by the operation or the information may
 * assist in correlating the response to the corresponding request
 * operation when multiple operations are requested by the application.
 */
typedef struct {
        NPF_IPv6UC_ReturnCode_t                 returnCode;
        union {
                NPF_IPv6UC_PrefixTableId_t      prefixTableId;
                NPF_IPv6UC_NextHopTableId_t     nextHopTableId;
                NPF_IPv6UC_FibTableId_t         fibTableId;
                NPF_IPv6UC_AddResTableId_t      addResTableId;
                NPF_uint32_t                    unused;
        } u1;
        union {
                NPF_IPv6UC_PfxCreateResp_t      prefixTableHandles;
                NPF_IPv6UC_Prefix_t             prefix;
                NPF_IPv6UC_PrefixQueryResp_t    prefixQueryResult;
                NPF_IPv6UC_NextHopTableHandle_t nextHopTableHandle;
                NPF_uint32_t                    nextHopIdentifier;
                NPF_IPv6UC_NextHopQueryResp_t   nextHopQueryResult;
                NPF_IPv6UC_FibCreateResp_t      fibTableHandles;
                NPF_IPv6UC_Prefix_t             fibPrefix;
                NPF_IPv6UC_FibQueryResp_t       fibQueryResult;
                NPF_IPv6UC_AddResTableHandle_t  addResTableHandle;
                NPF_IPv6UC_AddResKey_t          addResKey;
                NPF_IPv6UC_AddResQueryResp_t    addResQueryResult;
                NPF_uint32_t                    tableSpaceRemaining;
                NPF_uint32_t                    unused;
        } u2;
} NPF_IPv6UC_AsyncResponse_t;

/*
```

```
 * This structure is passed to the application as a parameter on a registered
 * completion callback. The type field indicates which function invocation
 * led to this response. The other three fields contain values depending
 * upon the invoking function, whether or not a single operation was
 * requested and whether the operations were successful or not.
 *
 * There are several possibilities:
 *
 * The application invokes a function requesting a single operation:
 *    - If allOK = TRUE, then numResp = 0 and the "resp" pointer is NULL.
 *      This indicates the operation completed successfully and there is
 *      no other additional response data to return.
 *    - If allOK = FALSE, then numResp = 1 and the "resp" pointer points to
 *      a response structure. If the returnCode field indicates NPF_NO_ERROR,
 *      the operation completed successfully and there is additional response
 *      data in the structure. Otherwise, the operation failed and the reason
 *      is indicated by the returnCode.
 * The application invokes a function requesting multiple operations:
 *    - If all operations completed successfully at the same time and there
 *      is no additional response data to provide, then allOK = TRUE,
 *      numResp = 0 and the "resp" pointer is NULL.
 *    - If all operations completed successfully at the same time, but there
 *      is additional response data to provide, then allOK = FALSE, numResp
 *      indicates the total number of requested operations and the "resp"
 *      pointer points to an array of response structures. The returnCode
 *      field will indicate NPF_NO_ERROR.
 *    - If some operations completed, but not all, then:
 *      > allOK = FALSE, numResp = the number of request operations
 *        completed.
 *      > The "resp" pointer will point to an array of response structures,
 *        each one containing one element for each completed request. For
 *        operations that completed successfully, the returnCode field will
 *        indicate NPF_NO_ERROR and additional response data may be present,
 *        depending on the type of function invocation. For operations that
 *        failed, the reason is indicated by the returnCode field.
 */

typedef struct {
    NPF_IPv6UC_CallbackType_t     type;
    NPF_boolean_t                 allOK;
    NPF_uint32_t                  numResp;
    NPF_IPv6UC_AsyncResponse_t   *resp;
} NPF_IPv6UC_CallbackData_t;

/*----------------------------------------------------------------
 *
 * Event Notification Data Types
 *
 *----------------------------------------------------------------*/

/*
 * Event Notification Types
 */
typedef enum NPF_IPv6UC_Event {
        NPF_IPV6UC_PREFIX_TBL_MISS        =  1,
        NPF_IPV6UC_NEXT_HOP_TBL_MISS      =  2,
        NPF_IPV6UC_FIB_PREFIX_MISS        =  4,
        NPF_IPV6UC_FWDTBL_REFRESH         =  5,
        NPF_IPV6UC_ADD_RES_TRANSITION     =  6
} NPF_IPv6UC_Event_t;

/* This event is triggered when the forwarding plane is unable to find a */
/* next hop identifier for a specific prefix. This event is optional.    */
typedef struct {
        NPF_IPv6UC_PrefixTableHandle_t   pfxTableHandle;
        NPF_IPv6Address_t                destIP_Address;
} NPF_IPv6UC_PrefixTblMiss_t;

/* This event is triggered when the forwarding plane is unable to find a */
/* next hop table entry for a specific next hop identifier. This event   */
/* is optional.                                                          */
typedef struct {
        NPF_IPv6UC_NextHopTableHandle_t  nextHopTableHandle;
        NPF_uint32_t                     nextHopIdentifier;
} NPF_IPv6UC_NextHopTblMiss_t;

/* This event is triggered when the forwarding plane is unable to find a */
/* FIB table entry for a specific IP address. This event is optional     */
typedef struct {
        NPF_IPv6UC_FibTableHandle_t      fibTableHandle;
        NPF_IPv6Address_t                destIP_Address;
} NPF_IPv6UC_FIB_PrefixMiss_t;

/*
 * This structure defines the enumerations for the table type used in
 * the NPF_IPv6UC_FwdTbl_Refresh_t structure below.
 */
typedef enum NPF_IPv6UC_TableType {
   NPF_IPV6UC_FIB_TABLE        = 1,
   NPF_IPV6UC_PREFIX_TABLE     = 2
} NPF_IPv6UC_TableType_t;

/* This event is triggered when the application or the IPv6 API          */
/* implementation needs to be notified that a FIB needs to be refreshed  */
/* on the forwarding plane. This event is optional.                      */
typedef struct {
        NPF_IPv6UC_TableType_t                  tableHandleType;
        union {
                NPF_IPv6UC_FibTableHandle_t     fibTableHandle;
                NPF_IPv6UC_PrefixTableHandle_t  prefixTableHandle;
        } u;
} NPF_IPv6UC_FwdTbl_Refresh_t;

/*
 * This structure defines the enumerations for the table type used in
 * the NPF_IPv6UC_AddResTransition_t structure below.
 */
typedef enum NPF_IPv6UC_AddResTableType {
   NPF_IPV6UC_ADD_RES_ADDRES_TABLE        = 1,
   NPF_IPV6UC_ADD_RES_FIB_TABLE           = 2,
   NPF_IPV6UC_ADD_RES_NEXTHOP_TABLE       = 3
} NPF_IPv6UC_AddResTableType_t;

/*
```

```
 * This event is triggered when the forwarding plane performs a
 * transition from one state of an address resolution entry to another
 * state. This event is optional.
 */
typedef struct {
   NPF_IPv6UC_AddResTableType_t                 addResTableType;
   union {
           NPF_IPv6UC_AddResTableHandle_t       addResTableHandle;
           NPF_IPv6UC_FibTableHandle_t          fibTableHandle;
           NPF_IPv6UC_NextHopTableHandle_t      nextHopTableHandle;
   } u;
   NPF_IPv6UC_AddResEntry_t                     addResEntry;
   NPF_IPv6_Reachability_t                      previousReachability;
} NPF_IPv6UC_AddResTransition_t;

/*
 * Event Notification Structures
 */
typedef struct {
        NPF_IPv6UC_Event_t                      type;
        union {
                NPF_IPv6UC_PrefixTableId_t      prefixTableId;
                NPF_IPv6UC_NextHopTableId_t     nextHopTableId;
                NPF_IPv6UC_FibTableId_t         fibTableId;
                NPF_IPv6UC_AddResTableId_t      addResTableId;
                NPF_uint32_t                    unused;
        } u1;
        union {
                NPF_IPv6UC_PrefixTblMiss_t      prefixTblMiss;
                NPF_IPv6UC_NextHopTblMiss_t     nextHopTblMiss;
                NPF_IPv6UC_FIB_PrefixMiss_t     fibPrefixMiss;
                NPF_IPv6UC_FwdTbl_Refresh_t     fwdTableRefreshRequest;
                NPF_IPv6UC_AddResTransition_t   addResTransition;
        } u2;
} NPF_IPv6UC_EventData_t;

/*
 * This structure is provided when the event notification handler
 * is invoked. It specifies one or more IPv6 unicast forwarding events.
 */

typedef struct {
        NPF_uint32_t                     numEvents;
        NPF_IPv6UC_EventData_t          *eventArray;
} NPF_IPv6UC_EventArray_t;

/*
 * Definitions for selectively enabling IPV6UC events
 */
#define NPF_IPV6UC_EV_PREFIX_TBL_MISS_ENABLE    (1 << 0)
#define NPF_IPV6UC_EV_NEXT_HOP_TBL_MISS_ENABLE  (1 << 1)
#define NPF_IPV6UC_EV_FIB_PREFIX_MISS_ENABLE    (1 << 2)
#define NPF_IPV6UC_EV_FWDTBL_REFRESH_ENABLE     (1 << 3)
#define NPF_IPV6UC_EV_ADD_RES_TRANSITION_ENABLE (1 << 4)
#define NPF_IPV6UC_EV_LAST                      (1 << 4)

/*-------------------------------------------------------------------
```

```
 *
 * Function Call Prototypes
 *
 *-----------------------------------------------------------------*/
typedef void (*NPF_IPv6UC_CallbackFunc_t) (
        NPF_IN NPF_userContext_t                userContext,
        NPF_IN NPF_correlator_t                 correlator,
        NPF_IN NPF_IPv6UC_CallbackData_t        data);

typedef void (*NPF_IPv6UC_EventCallFunc_t) (
        NPF_IN NPF_userContext_t                userContext,
        NPF_IN NPF_IPv6UC_EventArray_t          data);

NPF_error_t NPF_IPv6UC_Register(
        NPF_IN NPF_userContext_t                 userContext,
        NPF_IN NPF_IPv6UC_CallbackFunc_t         callbackFunc,
        NPF_OUT NPF_callbackHandle_t            *callbackHandle);

NPF_error_t NPF_IPv6UC_Deregister(
        NPF_IN NPF_callbackHandle_t              callbackHandle);

NPF_error_t NPF_IPv6UC_EventRegister(
        NPF_IN NPF_userContext_t                userContext,
        NPF_IN NPF_IPv6UC_EventCallFunc_t       eventCallFunc,
        NPF_IN NPF_eventMask_t                  eventMask,
        NPF_OUT NPF_callbackHandle_t           *eventCallHandle);

NPF_error_t NPF_IPv6UC_EventDeregister(
        NPF_IN NPF_callbackHandle_t              eventCallHandle);

NPF_IPv6UC_SupportedMode_t NPF_IPv6UC_GetSupportedModes (void);

NPF_IPv6UC_PreferredMode_t NPF_IPv6UC_GetPreferredMode(void);

NPF_error_t NPF_IPv6UC_PrefixTableHandleCreate(
        NPF_IN NPF_callbackHandle_t       callbackHandle,
        NPF_IN NPF_correlator_t           correlator,
        NPF_IN NPF_errorReporting_t       errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableId_t prefixTableId);

NPF_error_t NPF_IPv6UC_PrefixTableHandleDelete(
        NPF_IN NPF_callbackHandle_t                       callbackHandle,
        NPF_IN NPF_correlator_t                           correlator,
        NPF_IN NPF_errorReporting_t                       errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t             tableHandle);

NPF_error_t NPF_IPv6UC_PrefixEntryAdd(
        NPF_IN NPF_callbackHandle_t                       callbackHandle,
        NPF_IN NPF_correlator_t                           correlator,
        NPF_IN NPF_errorReporting_t                       errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t             tableHandle,
        NPF_IN NPF_uint32_t                               numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t                       *prefixArray,
        NPF_IN NPF_uint32_t                              *nextHopIdArray);

NPF_error_t NPF_IPv6UC_PrefixEntryDelete(
        NPF_IN NPF_callbackHandle_t             callbackHandle,
        NPF_IN NPF_correlator_t                 correlator,
        NPF_IN NPF_errorReporting_t             errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t   tableHandle,
        NPF_IN NPF_uint32_t                     numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t             *prefixArray);

NPF_error_t NPF_IPv6UC_PrefixTableFlush(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t           tableHandle);

NPF_error_t NPF_IPv6UC_PrefixTableAttributeQuery(
        NPF_IN NPF_callbackHandle_t                      callbackHandle,
        NPF_IN NPF_correlator_t                          correlator,
        NPF_IN NPF_errorReporting_t                      errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t            tableHandle);

NPF_error_t NPF_IPv6UC_PrefixEntryQuery(
        NPF_IN NPF_callbackHandle_t              callbackHandle,
        NPF_IN NPF_correlator_t                  correlator,
        NPF_IN NPF_errorReporting_t              errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t    tableHandle,
        NPF_IN NPF_uint32_t                      numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t              *prefixArray);

NPF_error_t NPF_IPv6UC_PrefixNextHopTableBind(
        NPF_IN NPF_callbackHandle_t              callbackHandle,
        NPF_IN NPF_correlator_t                  correlator,
        NPF_IN NPF_errorReporting_t              errorReporting,
        NPF_IN NPF_IPv6UC_PrefixTableHandle_t    prefixTableHandle,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t   nextHopTableHandle);


NPF_error_t NPF_IPv6UC_NextHopTableHandleCreate(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableId_t              nextHopTableId);

NPF_error_t NPF_IPv6UC_NextHopTableHandleDelete(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t          tableHandle);

NPF_error_t NPF_IPv6UC_NextHopEntryAdd(
        NPF_IN NPF_callbackHandle_t                      callbackHandle,
        NPF_IN NPF_correlator_t                          correlator,
        NPF_IN NPF_errorReporting_t                      errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t           tableHandle,
        NPF_IN NPF_uint32_t                              numEntries,
        NPF_IN NPF_uint32_t                             *nextHopIdArray,
        NPF_IN NPF_IPv6UC_NextHopArray_t                *nextHopArrays);

NPF_error_t NPF_IPv6UC_NextHopEntryDelete(
        NPF_IN NPF_callbackHandle_t                      callbackHandle,
        NPF_IN NPF_correlator_t                          correlator,
        NPF_IN NPF_errorReporting_t                      errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t           tableHandle,
        NPF_IN NPF_uint32_t                              numEntries,
        NPF_IN NPF_uint32_t                             *nextHopIdArray);

NPF_error_t NPF_IPv6UC_NextHopTableFlush(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t          tableHandle);

NPF_error_t NPF_IPv6UC_NextHopTableAttributeQuery(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t          tableHandle);

NPF_error_t NPF_IPv6UC_NextHopEntryQuery(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_NextHopTableHandle_t          tableHandle,
        NPF_IN NPF_uint32_t                             numEntries,
        NPF_IN NPF_uint32_t                            *nextHopIdArray);

NPF_error_t NPF_IPv6UC_FibTableHandleCreate(
        NPF_IN NPF_callbackHandle_t             callbackHandle,
        NPF_IN NPF_correlator_t                 correlator,
        NPF_IN NPF_errorReporting_t             errorReporting,
        NPF_IN NPF_IPv6UC_FibTableId_t          fibTableId);

NPF_error_t NPF_IPv6UC_FibTableHandleDelete(
        NPF_IN NPF_callbackHandle_t           callbackHandle,
        NPF_IN NPF_correlator_t               correlator,
        NPF_IN NPF_errorReporting_t           errorReporting,
        NPF_IN NPF_IPv6UC_FibTableHandle_t    tableHandle);

NPF_error_t NPF_IPv6UC_FibEntryAdd(
        NPF_IN NPF_callbackHandle_t              callbackHandle,
        NPF_IN NPF_correlator_t                  correlator,
        NPF_IN NPF_errorReporting_t              errorReporting,
        NPF_IN NPF_IPv6UC_FibTableHandle_t       tableHandle,
        NPF_IN NPF_uint32_t                      numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t              *prefixArray,
        NPF_IN NPF_IPv6UC_NextHopArray_t        *nextHopArrays);

NPF_error_t NPF_IPv6UC_FibEntryDelete(
        NPF_IN NPF_callbackHandle_t              callbackHandle,
        NPF_IN NPF_correlator_t                  correlator,
        NPF_IN NPF_errorReporting_t              errorReporting,
        NPF_IN NPF_IPv6UC_FibTableHandle_t       tableHandle,
        NPF_IN NPF_uint32_t                      numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t              *prefixArray);

NPF_error_t NPF_IPv6UC_FibTableFlush(
        NPF_IN NPF_callbackHandle_t              callbackHandle,
        NPF_IN NPF_correlator_t                  correlator,
        NPF_IN NPF_errorReporting_t              errorReporting,
        NPF_IN NPF_IPv6UC_FibTableHandle_t       tableHandle);

NPF_error_t NPF_IPv6UC_FibTableAttributeQuery(
        NPF_IN NPF_callbackHandle_t         callbackHandle,
        NPF_IN NPF_correlator_t             correlator,
        NPF_IN NPF_errorReporting_t         errorReporting,
        NPF_IN NPF_IPv6UC_FibTableHandle_t  tableHandle);

NPF_error_t NPF_IPv6UC_FibEntryQuery(
        NPF_IN NPF_callbackHandle_t              callbackHandle,
        NPF_IN NPF_correlator_t                  correlator,
        NPF_IN NPF_errorReporting_t              errorReporting,
        NPF_IN NPF_IPv6UC_FibTableHandle_t       tableHandle,
        NPF_IN NPF_uint32_t                      numEntries,
        NPF_IN NPF_IPv6UC_Prefix_t              *prefixArray);


NPF_error_t NPF_IPv6UC_AddResTableHandleCreate(
        NPF_IN NPF_callbackHandle_t         callbackHandle,
        NPF_IN NPF_correlator_t             correlator,
        NPF_IN NPF_errorReporting_t         errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableId_t   addResTableId);

NPF_error_t NPF_IPv6UC_AddResTableHandleDelete(
        NPF_IN NPF_callbackHandle_t                    callbackHandle,
        NPF_IN NPF_correlator_t                        correlator,
        NPF_IN NPF_errorReporting_t                    errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableHandle_t          tableHandle);

NPF_error_t NPF_IPv6UC_AddResEntryAdd(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableHandle_t           tableHandle,
        NPF_IN NPF_uint32_t                             numEntries,
        NPF_IN NPF_IPv6UC_AddResEntry_t                *entryArray);

NPF_error_t NPF_IPv6UC_AddResEntryDelete(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableHandle_t           tableHandle,
        NPF_IN NPF_uint32_t                             numEntries,
        NPF_IN NPF_IPv6UC_AddResKey_t                  *entryArray);

NPF_error_t NPF_IPv6UC_AddResTableFlush(
        NPF_IN NPF_callbackHandle_t                    callbackHandle,
        NPF_IN NPF_correlator_t                        correlator,
        NPF_IN NPF_errorReporting_t                    errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableHandle_t          tableHandle);

NPF_error_t NPF_IPv6UC_AddResAttributeQuery(
        NPF_IN NPF_callbackHandle_t                    callbackHandle,
        NPF_IN NPF_correlator_t                        correlator,
        NPF_IN NPF_errorReporting_t                    errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableHandle_t           tableHandle);

NPF_error_t NPF_IPv6UC_AddResEntryQuery(
        NPF_IN NPF_callbackHandle_t                     callbackHandle,
        NPF_IN NPF_correlator_t                         correlator,
        NPF_IN NPF_errorReporting_t                     errorReporting,
        NPF_IN NPF_IPv6UC_AddResTableHandle_t           tableHandle,
        NPF_IN NPF_uint32_t                             numEntries,
        NPF_IN NPF_IPv6UC_AddResKey_t                  *entryArray);


#ifdef __cplusplus
}
#endif

#endif /* __NPF_IPV6U_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		