Interface Management API Implementation Agreement (Core Function Set)

Revision 3.0

Editor: John Renwick, Agere Systems, jrenwick@agere.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 1.0.

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
3.0	11/22/2004	Created Rev 0.0 from the Interface Management IA version 2.0 by removing all type-specific definitions, leaving the generic Interface Management Core definitions and functions. Modified definitions extensively to move generic features to the Core document and make Interface Management header files independent. Many of these changes are not backward-compatible with version 2.0.

2 Introduction

A network element, for instance a router, has one or more physical connection points, usually called *links,* through which it is connected to other network elements. Packets are received over a link by the network element for processing. A link usually has an associated Layer 2 (L2) protocol that is used to transfer packets over the media of the link. A L2 protocol typically either implements and/or negotiates standards-based link characteristics such as link speed, single or full duplex transmission mode, etc. PPP, Ethernet, etc. are examples of layer 2 protocols commonly deployed in today's networks. It is quite possible that more than one L2 protocol can be running on a single link, e.g. PPP over Ethernet. Also, multiple L2 interfaces of the same type can be combined into a single logical L2 interface to create a trunk, as in 802.3ad link aggregation and other multilink techniques.

The figures below show some of the relationships that exist in today's networks. Other configurations are possible, including combining the forms below into more deeply nested hierarchies.

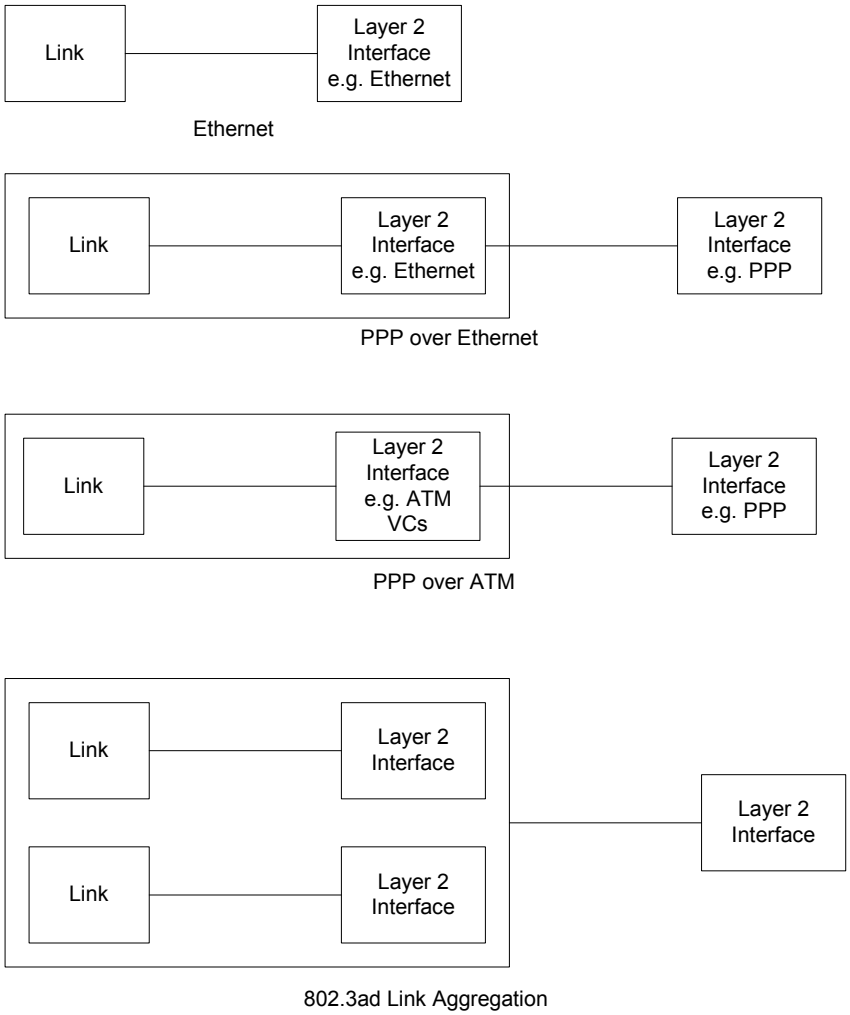

Figure 1 - Layer 2 interface configurations

One or more Layer 3 protocols, for instance IPv4, IPv6 or IPX, can be used on an L2 interface. An L3 interface captures the properties of the corresponding L3 protocol. For example, in case of IPv4, IP address and prefix length are associated with the L3 interfaces.

Figure 2 - Layer 2 and Layer 3 Interface Relationship

There is a many-to-many relationship between L2 and L3 interfaces. Thus, as shown by Figure 3, multiple Layer 3 interfaces can be associated with a single L2 interface, and a single L3 interface can be associated with multiple L2 interfaces.

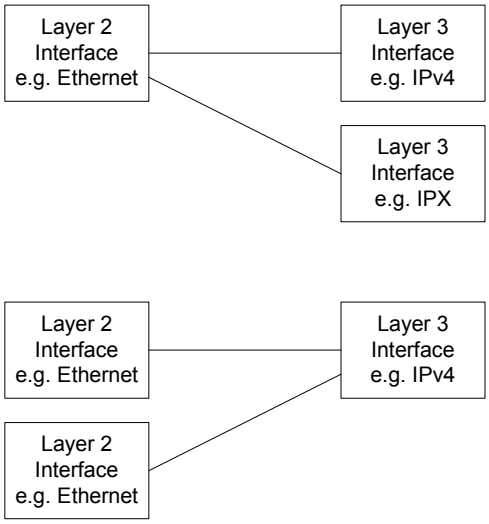

Figure 3 - L2-L3 mappings

Figure 4 shows the overall relationships between links, L2 Interfaces and L3 Interfaces.

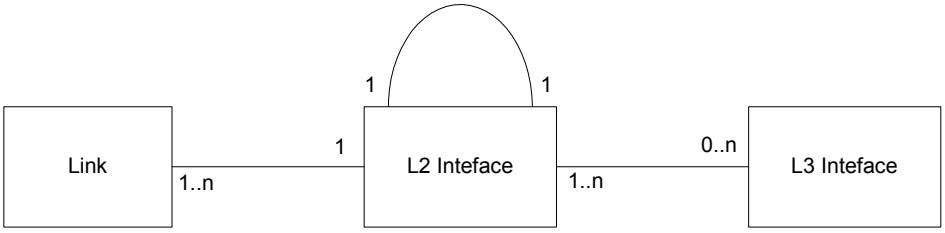

Figure 4 - Link/L2/L3 relationship

The Interface Management API provides a uniform interface for configuring and managing the physical and logical interfaces of which a network processor may need to be aware. For example, the API defined in this document will cover aspects of interface management related to Layer 2 (Bridging), Layer 3 (IP), media-specific management (Ethernet, ATM UNI, SONET, etc.), and so on.

2.1 Assumptions and External Requirements

1. Memory allocation and usage model for the API implementation will be as dictated by the NPF Software Conventions Implementation Agreement.
2. The API does not determine any policy with respect to operations on interfaces or their events. It is assumed that policy will be embodied in an Interface Manager module that is part of the application. (See 2.4.4.)

2.2 Scope

The "interfaces" addressed by this API are those related to external network ports only. Other internal interfaces defined by NP Forum, such as streaming and lookaside interfaces, are outside the scope of this document.

This document describes only the "core" functionality of Interface Management; that is, definitions and functions that apply to any interface, regardless of its type (such as LAN, POS, IPv4, etc.). Other IAs exist to define APIs related to the specifics of each interface type.

Every implementation of Interface Management must support the required definitions and functions of the Core document. Some implementations will support only a subset of the defined types, and for such implementations only the required API features in the documents specific to those types need be supported. There are type-specific documents such as IPv4 and IPv6 that define support for multiple interface types. In those cases, a vendor can claim support for and implement a subset of the definitions found in the appropriate type-specific document.

2.3 Dependencies

This document depends on the NPF Software Implementation Agreement – Software API Conventions, Revision 2, for basic type definitions.

2.4 Interface Management Structures

We represent an interface with a hierarchy of structures. At the top level is a structure containing attributes that can be set from the application, and are common to all interface types. Within that is a union containing objects that are attributes of a specific type or family of interface types. This nested structure contains only attributes that can be set on an interface. There are other structures for reading interface attributes, such as statistics. See the individual structure descriptions below.

2.4.1 Common Interface Attributes

The common interface attribute structure contains the following, which the application can set on most interfaces:

- Interface type code, which indicates which of several different type-specific groups of attributes are being used: LAN, IPv4, ATM UNI, POS, and so on.
- Administrative Status (up or down): a global enable/disable control on the interface.
- Operational Status (up or down): whether or not the interface and/or its children are in working condition.
- Interface Identifier: a nonzero integer value assigned by the application to each interface it creates. No two interfaces may have the same ID. This number can be anything of the application's choosing; the value of ifIndex (see RFC 2863) is one possiblity.
- Maximum PDU size: the largest Protocol Data Unit (PDU) that can be transmitted. The actual packet sent might be larger, due to headers being added by processes and hardware that might be represented by parents of this interface.
- Forwarding Mode: for interface types that support forwarding of packets or cells, forwarding can be enabled or disabled. Disabling forwarding does not prevent delivery of locally-addressed packets.
- Loopback modes: for interfaces that support some kind of loopback for testing or diagnostic purposes, there are common attributes and functions to control it.

2.4.2 Common Interface Statistics

These attributes have a structure of their own, and can be retrieved by an application, but not set. They apply to all interface types. These counters correspond to counters defined in MIB-II (RFC 1213) and the Interfaces Group MIB (RFC 2233).

- Counters (64 bits – wide enough "never"[1] to wrap):
  - Bytes received
  - Input packets (unicast)
  - Input packets (multicast)
  - Input packets (broadcast)
  - Input packets dropped
  - Input errors
  - Input packets of unknown protocol
  - Bytes sent
  - Output packets (unicast)
  - Output packets (multicast)
  - Output packets (broadcast)
  - Output packets dropped
  - Output errors

2.4.3 Interface Relatedness

The API includes a function (NPF_IfBind) that relates a pair of interfaces as parent and child. A parent interface represents a lower layer than that of its children, with reference to the OSI model. For example, layer 2 interfaces are naturally parents layer 3 interfaces. An interface can be at the same time a parent of one interface and a child of another. The API places no restriction on:

- the number of levels of hierarchy
- the numer of child interfaces any can have
- the number of parent interfaces any can have
- the types of interfaces that can be bound together as parent and child.

This last point means, for instance, that binding a LAN interface as the child of an IPv4 interface is permitted as far as the API specification is concerned, even though such a binding might make no sense in the context of a given implementation (for another implementation, it might make perfect sense). Implementations MAY place their own restrictions on the way interfaces of certain types can be related, in what multiplicity, and to what depth of hierarchy.

2.4.4 Interface Manager Application

Because interfaces can be related, an application may require that an event on one interface causes a related event to be registered on a related interface; or it may require that operations on related interfaces be done in a certain way, or in a certain order. The API imposes a few necessary restrictions on the order of operations (see section 4.4), but there are other matters of policy that belong to the application and are outside the scope of the API to regulate. Where there are significant policy considerations, the client application should include an Interface Manager module that brokers transactions or intercedes between the Interface Management API and its clients, and ensures that the application's requirements are satisfied.

[1] At OC768 sustained full speed, or 39813 megabits/second, a 64-bit byte counter will wrap in approximately 117 years.

3 Data Types

3.1 Interface Management API Types

3.1.1 Interface Identifier

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

```
typedef NPF_uint32_t    NPF_IfID_t;       /* Interface Identifier */
```

3.1.2 Generic Interface Structure: NPF_IfGeneric_t

This structure contains the "generic" attributes of an interface – that is, the attributes that are common to all interface types. It also may contain a pointer to a type-specific structure. As such, this structure can carry all attributes of an interface (not including counter values). It is used in functions that set interface attributes and query interface attributes.

The **`NPF_IfGeneric_t`** structure contains forward references to interface-type-specific structures that are defined in other header files. These references are pointers contained in a union within the **`NPF_IfGeneric_t`** structure. For each interface type, only one type-specific attribute structure is defined. Each implementer will need to customize the union to include only pointers as needed for the interface types supported.

```
/*
 *    The Interface structure:
 */
typedef struct  {
      NPF_IfID_t        ifID;       /* Interface ID */
      NPF_IfType_t      type;       /* Logical interface type */
      NPF_uint64_t      speed;      /* Speed in Kbits/second */
      NPF_uint32_t        maxPDU;   /* Max Protocol Data Unit Size */
      NPF_IfOperStatus_t operStatus; /* Operational Status (read only)*/
      NPF_IfAdminStatus_t adminStatus; /* Administrative up/down */
      NPF_IfFwdMode_t   fwdMode;          /* Forwarding Mode */
      NPF_IfInternalLoopbackMode_t  intLoop;    /* Internal loopback */
      NPF_IfExternalLoopbackMode_t  extLoop;    /* External loopback */
      NPF_uint32_t      nChildren;  /* Number of child interfaces */
      NPF_uint32_t      *childIDs;  /* Array of child interface IDs */
      NPF_uint32_t      nParents;     /* Number of parent interfaces */
      NPF_uint32_t      *parentIDs;   /* Array of parent i/f IDs */
      union   {         /* Type specific attributes (by if_type code) */

      /* **** CAUTION ****
       * ONLY POINTERS TO STRUCTURES MAY BE USED IN THIS UNION.
```

```
       * **** CAUTION **** */

      /* The implementer adds lines like the following,
       * depending on the interface types supported. In
       * this example, we have support for LAN and IPv4
       * interface types.
       */
          NPF_IfLAN_t *LAN_Attr;        /* LAN interface attributes */
          NPF_IfIPv4_t *IPv4_Attr;      /* IPv4 Interface attributes */
      } u;
} NPF_IfGeneric_t;
```

The Operational Status variable is read-only. It reflects the mechanical and electrical status and software readiness of this and underlying interfaces, and cannot be set by the application.

The arrays of parent and child interface IDs are read-only. **`NPF_IfBind()`** and **`NPF_IfUnBind()`** must be used to create or modify interface bindings. These functions take interface handles as arguments. When an application asks for interface settings using **`NPF_IfAttrGet()`**, the implementation maps the handles to Interface IDs and returns the ID values in these arrays.

3.1.3 Interface Handle: NPF_IfHandle_t

```
/*
 * Interface handle
 */
typedef NPF_uint32_t    NPF_IfHandle_t;
```

The following values are reserved and MUST not be assigned to any valid interface handle.

```
#define  NPF_IF_HANDLE_NULL 0             /* NULL handle value */
#define  NPF_IF_HANDLE_ALL  0xFFFFFFFF    /* Represents all interfaces */
```

3.1.4 Interface Type Code: NPF_IfType_t

The interface type code identifies the type of interface in the NPF_IfGeneric_t structure and other places. It is also used as a qualifier for various other type-specific codes defined in other Implementation Agreements. The general convention is to define type-specific codes as follows:

```
#define NPF_XXX_CODE1 ((NPF_IF_TYPE_XXX<<16) + code)
```

where "XXX" is the name of the interface type, and "code" is a numeric value. This style will be used in type-specific IAs to define function type codes and event type codes. Error codes defined in type-specific documents will have values assigned similarly, but with a slight difference, using a macro:

```
#define NPF_IF_E_XXX_CODE(code) (0x10000+(NPF_IF_TYPE_XXX<<8)+(code))

#define NPF_IF_E_<reason> NPF_IF_E_XXX_CODE(<nn>)
```

The following interface types were defined at the time of this writing. Others may be defined in newer Interface Management Implementation Agreements; see the individual type-specific API documents for the complete set.

```
#define NPF_IF_TYPE_RESV            0           /* Reserved value */
```

```
#define NPF_IF_TYPE_UNK             1           /* Unknown interface type */
#define NPF_IF_TYPE_LAN             2           /* LAN interface */
#define NPF_IF_TYPE_ATM             3           /* ATM interface */
#define NPF_IF_TYPE_POS             4           /* Packet over SONET */
#define NPF_IF_TYPE_IPV4            5           /* IPv4 logical interface */
#define NPF_IF_TYPE_IPV6            6           /* IPv6 logical interface */

typedef NPF_uint32_t NPF_IfType_t;
```

3.1.5 Structure to Relate Two Interfaces: NPF_IfBinding_t

```
/*
 *    Structure to relate two interfaces
 */
typedef struct {
      NPF_IfHandle_t    parent;           /* Parent interface handle */
      NPF_IfHandle_t    child;            /* Child interface handle */
} NPF_IfBinding_t;
```

3.1.6 Interface Statistics: NPF_IfStatistics_t

```
/*
 *    Statistics
 */
typedef struct {
      NPF_uint64_t      bytesRx;          /* Receive Bytes */
      NPF_uint64_t      ucPackRx;         /* Receive Unicast Packets */
      NPF_uint64_t      mcPackRx;         /* Receive Multicast Packets */
      NPF_uint64_t      bcPackRx;         /* Receive Broadcast Packets */
      NPF_uint32_t      dropRx;           /* Receive packets dropped */
      NPF_uint32_t      errorRx;          /* Receive errors */
      NPF_uint32_t      protoRx;          /* Receive unknown protocol */
      NPF_uint64_t      bytesTx;          /* Transmit bytes */
      NPF_uint64_t      ucPackTx;         /* Transmit Unicast Packets */
      NPF_uint64_t      mcPackTx;         /* Transmit Multicast Packets */
      NPF_uint64_t      bcPackTx;         /* Transmit Broadcast Packets */
      NPF_uint32_t      dropTx;           /* Transmit dropped packets */
      NPF_uint32_t      errorTx;          /* Transmit errors */
} NPF_IfStatistics_t;
```

3.1.7 Operational Status Code: NPF_IfOperStatus_t

NPF_IfOperStatus_t is meant to mirror the ifOperStatus object in the Interfaces Group MIB (RFC 2863). Please refer to that document, section 3.1.13, for details on the meaning and behavior of these states.

```
/*
 *    Operational Status code
 */
typedef enum      {
      NPF_IF_OPER_STATUS_UP = 1,          /* Operationally UP */
      NPF_IF_OPER_STATUS_DOWN = 2,        /* Operationally DOWN */
      NPF_IF_OPER_STATUS_TESTING = 3,     /* Testing status */
      NPF_IF_OPER_STATUS_UNKNOWN = 4,     /* Status unknown */
```

```
      NPF_IF_OPER_STATUS_DORMANT = 5,      /* Dormant status */
      NPF_IF_OPER_STATUS_NOT_PRESENT = 6, /* Interface not present */
      NPF_IF_OPER_STATUS_LOWER_LAYER_DOWN = 7 /* Parent I/F down */
} NPF_IfOperStatus_t;
```

3.1.8 Administrative Status Code: NPF_IfAdminStatus_t

NPF_IfAdminStatus_t is meant to mirror the ifAdminStatus object in the Interfaces Group MIB (RFC 2863). Please refer to that document, section 3.1.13, for details on the meaning and behavior of these states.

```
/*
 *    Administrative Status code
 */
typedef enum      {
      NPF_IF_ADMIN_STATUS_UP = 1,          /* Administratively UP */
      NPF_IF_ADMIN_STATUS_DOWN = 2,        /* Administratively DOWN */
      NPF_IF_ADMIN_STATUS_TESTING = 3      /* Testing status */
} NPF_IfAdminStatus_t;
```

3.1.9 Forwarding Mode : NPF_IfFwdMode_t

Forwarding is a function defined for several interface types, including IPv4, IPv6, and LAN interfaces. The Forwarding Mode variable has meaning on these interface types, but not on those (such as POS) for which no forwarding function is defined.

```
/*
 *    Forwarding mode code
 */
typedef enum      {
      NPF_IF_FORWARDING_ENABLE = 1, /* Enable Forwarding */
      NPF_IF_FORWARDING_DISABLE = 2 /* Disable Forwarding */
} NPF_IfFwdMode_t;
```

3.1.10 Loopback Modes

Many interface types support a loopback function, either by hardware or software. External Loopback means that packets received from outside are turned around and sent back to their source. Internal loopback means that packets sent from the local system to a remote system, instead of being sent to their destinations, are directed back to the local system.
Support of loopback in any particular interface type is optional. Since the primary use of loopback is to help in isolating faults in the data path, implementations that support both internal and external loopback SHOULD select an external loopback point closer to the external interface than the internal loopback point, so as to minimize the likelihood of a single fault causing both loopbacks to fail.

```
/*
 * Internal and External Loopback Modes
 */
typedef     enum  {
      NPF_IF_INTERNAL_LOOPBACK_ENABLE     = 1,  /* Enable loopback */
      NPF_IF_INTERNAL_LOOPBACK_DISABLE    = 2,  /* Disable loopback */
```

```
} NPF_IfInternalLoopbackMode_t;

typedef     enum  {
      NPF_IF_EXTERNAL_LOOPBACK_ENABLE     = 1,  /* Enable loopback */
      NPF_IF_EXTERNAL_LOOPBACK_DISABLE    = 2,  /* Disable loopback */
} NPF_IfExternalLoopbackMode_t;
```

3.1.11 Interface Identity : NPF_IfIdentity_t

This structure, and the Interface Identity Array following this one, are used for the response from the **`NPF_IfHandleGetAll()`** function call.

```
/*
 *    Interface Identity (ID and Handle)
 */
typedef struct {
      NPF_IfID_t     ifID;
      NPF_IfHandle_t ifHandle;
} NPF_IfIdentity_t;
```

3.1.12 Interface Identity Array : NPF_IfIdentityArray_t

```
/*
 *    Interface Identity Array
 */
typedef struct {
      NPF_uint32_t nCount;
      NPF_IfIdentity_t *ifIdentityArray;
} NPF_IfIdentityArray_t;
```

3.1.13 Binding Update Information

```
/*
 *    Binding change type
 */
typedef enum     {
     IF_BIND_ADD     = 0,  /* add parent-child relationship */
     IF_BIND_DELETE  = 1   /* delete parent-child relationship */
} NPF_IfBindAction_Type_t;

/*
 *    Parent-child Interface relationship changes
 *
 */
typedef struct
{
      NPF_IfBindAction_Type_t bindChangeType; /* delete or add relationship*/
      NPF_IfBinding_t         ifBind;
} NPF_IfBind_Update_t;
```

3.1.14 Address Update Information

These data structures support the event notifications triggered for L3 address changes.

```
/*
 * Action for address changes
```

```
 */
typedef enum     {
     IF_ADDR_ADD     = 0,
     IF_ADDR_DELETE  = 1,
     IF_ADDR_MODIFY   = 2
} NPF_IfAddrUpdate_Type_t;

/*
 *    L2/L3 Address type
 */
typedef enum     {
  IF_IPV4_ADDR   = 1,  /* modify primary IPv4 address */
  IF_IPV4_UCADDR = 2,  /* Add, delete IPv4 unicast addresses */
  IF_IPV4_MCADDR = 3,  /* Add, delete Multicast addresses */
  IF_IPV6_ADDR = 4,     /* Add, delete IPv6 addresses */
  IF_MAC_ADDR = 5       /* Add, delete MAC addresses */
} NPF_IfL2L3Addr_Type_t;


/*
 *    L2/L3 Address Changes
 */
typedef struct
{
   NPF_IfAddrUpdate_Type_t   addrChangeType; /* add, delete or modify */
   NPF_IfL2L3Addr_Type_t       addrType;     /* Ipv4, Ipv6, MAC addr, etc. */

   NPF_uint32_t          nAddrs
   union {
      NPF_IPv4Prefix_t   *if_IPv4AddrArray;
      NPF_IPv6Prefix_t   *if_IPv6AddrArray;
      NPF_IfMacAddress_t *macAddrArray;
   } newAddr;
} NPF_IfL2L3Addr_Update_t;
```

3.2 Data Structures for Completion Callbacks

3.2.1 Completion Callback Type (NPF_IfCallbackType_t)

These codes are used in asynchronous responses from API function calls. They tell the client which function is giving the response. The codes defined here are only those for the core functions defined in this document. Other API documents that define functions for specific interface types will also define codes to be used as values of this typedef. Those documents will qualify their codes using the applicable interface type (see 3.1.4) to ensure their callback type codes are unique.

```
typedef NPF_uint32_t NPF_IfCallbackType_t;

/*
 *    Completion Callback Types (generic set)
 */
#define NPF_IF_CREATE              1
#define NPF_IF_DELETE              2
#define NPF_IF_BIND                3
#define NPF_IF_UN_BIND             4
```

```
#define NPF_IF_STATS_GET            5
#define NPF_IF_ATTR_SET             6
#define NPF_IF_CREATE_AND_SET       7
#define NPF_IF_ENABLE               8
#define NPF_IF_DISABLE              9
#define NPF_IF_OPER_STATUS_GET      10
#define NPF_IF_MAX_PDU_SIZE_SET     11
#define NPF_IF_ATTR_GET             12
#define NPF_IF_FWD_ENABLE           13
#define NPF_IF_FWD_DISABLE          14
#define NPF_IF_INTERNAL_LOOPBACK_ENABLE   15
#define NPF_IF_INTERNAL_LOOPBACK_DISABLE  16
#define NPF_IF_EXTERNAL_LOOPBACK_ENABLE   17
#define NPF_IF_EXTERNAL_LOOPBACK_DISABLE  18
#define NPF_IF_HANDLE_GET           19
#define NPF_IF_HANDLE_GET_ALL       20
```

3.2.2 Asynchronous Response Array Element: NPF_IfAsyncResponse_t

```
/*
 *    An asynchronous response contains an interface handle,
 *    an error or success code, and in some cases a pointer to
 *    a function-specific structure embedded in a union.  One or
 *    more of these is passed to the callback function as an array
 *    within the NPF_IfCallbackData_t structure (below).
 */
typedef struct  {       /* Asynchronous Response Structure */
    NPF_IfHandle_t      ifHandle;   /* I/F handle for this response */
    NPF_IfID_t          ifID;       /* Interface ID */
    NPF_IfType_t        ifType;     /* Interface Type */
    NPF_IfErrorType_t error;        /* Error code for this response */
    union { /* Function-specific response information:      */

      /* **** CAUTION ****
       * EACH MEMBER OF THIS UNION MUST BE THE SAME SIZE,
       * EQUAL TO THE SIZE OF A POINTER VARIABLE.
       * **** CAUTION **** */

      /* For generic functions */
      NPF_uint32_t      unused;           /* Default */
      NPF_uint32_t      arrayIndex; /* NPF_IfCreateAndSet index */
      NPF_IfStatistics_t *ifStats;  /* NPF_IfGenericStatsGet() */
      NPF_IfOperStatus_t operStat;  /* NPF_IfOperStatusGet() */
      NPF_IfHandle_t    child;            /* NPF_IfBind(), handle=parent*/
      NPF_IfGeneric_t   *attrs;           /* NPF_IfAttrGet() */
      NPF_IfIdentifyArray_t *idArray      /* NPF_IfHandleGetAll() */

      /* For type-specific functions */

      /*
       * The implementer must add lines like the following,
       * as needed for the responses of functions supporting
       * the interface types included in the implementation.
       * In this example we have support for LAN and IPv4
       * interface types.  Each member must be the same
```

```
       * size: the size of a pointer variable.
       */

      NPF_MAC_Address_t *MACaddr;   /* NPF_IfLAN_SrcAddrGet() */
      NPF_IPv4Prefix_t  *v4prefix;  /* NPF_IPv4UC_AddrAdd(), */
                                    /* Set(),Delete() */
      NPF_IPv4Address_t *v4addr;    /* NPF_IPv4McastAddrAdd(), */
                                    /* Set() */
    } u;

} NPF_IfAsyncResponse_t;
```

The following table summarizes the information returned by each function in this API, pointed to by the "specific" variable in **`NPF_IfAsyncResponse_t`**.

Function Name	Type Code	Structure Returned
NPF_IfCreate	NPF_IF_CREATE	Unused (null pointer)
NPF_IfDelete	NPF_IF_DELETE	Unused (null pointer)
NPF_IfBind	NPF_IF_BIND	NPF_IfHandle_t (child)
NPF_IfUnBind	NPF_IF_UN_BIND	NPF_IfHandle_t (child)
NPF_IfGenericStatsGet	NPF_IF_STATS_GET	NPF_IfStatistics_t *
NPF_IfAttrSet	NPF_IF_ATTR_SET	Unused (null pointer)
NPF_IfCreateAndSet	NPF_IF_CREATE_AND_SET	NPF_uint32_t (arrayIndex)
NPF_IfEnable	NPF_IF_ENABLE	Unused (null pointer)
NPF_IfDisable	NPF_IF_DISABLE	Unused (null pointer)
NPF_IfOperStatusGet	NPF_IF_OPER_STATUS_GET	NPF_IfOperStatus_t
NPF_IfMaxPDU_SizeSet	NPF_IF_MAX_PDU_SIZE_SET	Unused (null pointer)
NPF_IfAttrGet	NPF_IF_ATTR_GET	NPF_IfGeneric_t *
NPF_IfFwdEnable	NPF_IF_FWD_ENABLE	Unused (null pointer)
NPF_IfFwdDisable	NPF_IF_FWD_DISABLE	Unused (null pointer)
NPF_IfInternalLoopbackEnable	NPF_IF_INTERNAL_LOOPBACK_ENABLE	Unused (null pointer)
NPF_IfInternalLoopbackDisable	NPF_IF_INTERNAL_LOOPBACK_DISABLE	Unused (null pointer)
NPF_IfExternalLoopbackEnable	NPF_IF_EXTERNAL_LOOPBACK_ENABLE	Unused (null pointer)
NPF_IfExternalLoopbackDisable	NPF_IF_EXTERNAL_LOOPBACK_DISABLE	Unused (null pointer)
NPF_IfHandleGet	NPF_IF_HANDLE_GET	Unused (null pointer)
NPF_IfHandleGetAll	NPF_IF_HANDLE_GET_ALL	NPF_IfIdentityArray *

3.2.3 Callback Data Structure: NPF_IfCallbackData_t

```
/*
 *    The callback function receives the following structure containing
 *    one or more asynchronous responses from a single function call.
 *    There are several possibilities:
 *    1. The called function does a single request
 *       - n_resp = 1, and the resp array has just one element.
 *       - allOK = TRUE if the request completed without error.
 *         and the only return value is the response code.
 *       - if allOK = FALSE, the "resp" structure has the error code.
 *    2. The called function supports an array of requests
 *       a. All completed successfully, at the same time, and the
 *          only returned value is the response code:
 *          - allOK = TRUE, n_resp = 0.
 *       b. Some completed, but not all, or there are values besides
```

```
 *          the response code to return:
 *          - allOK = FALSE, n_resp = the number completed.
 *          - the "resp" array will contain one element for
 *            each completed request, with the error code
 *            in the NPF_IfAsyncResponse_t structure, along
 *            with any other information needed to identify
 *            which request element the response belongs to.
 *          - 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.
 */
typedef struct {
      NPF_IfCallbackType_t    type;       /* Which function was called? */
      NPF_boolean_t           allOK;      /* TRUE if all completed OK */
      NPF_uint32_t            n_resp;     /* Number of responses in array */
      NPF_IfAsyncResponse_t  *resp;       /* Pointer to response structures*/
} NPF_IfCallbackData_t;
```

3.3 Error Codes (NPF_IfErrorType_t)

The codes defined here are generic Interface Management error codes that may apply to more than one type of interface. Additional Interface Management error codes are defined as values of the same (NPF_IfErrorType_t) typedef in the Interface Management Implementation Agreements related to specific interface types. These documents shall qualify their error code values using the applicable interface type code, as described in section 3.1.4, to guarantee the uniqueness of all Interface Management error codes.

N.B.: this is a departure from the original Software Conventions document, which assigned a range from 100 to 200 for all Interface Management error codes. Only the generic Interface Management error codes will follow that convention. Error codes defined in type-specific documents will have values assigned with an offset of 0x1tt00, where “tt” is the interface type code:

```
#define NPF_IF_E_XXX_CODE(code) (0x10000+(NPF_IF_TYPE_XXX<<8)+(code))

#define NPF_IF_E_<reason> NPF_IF_E_XXX_CODE(<nn>)


/*
 *    Error codes */

/* Callback/event reg. error */
/*****************************************************
 * Note: The following code is deprecated.           *
 * Use NPF_E_CALLBACK_ALREADY_REGISTERED instead.    *
 *****************************************************/
#define NPF_IF_E_ALREADY_REGISTERED ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR)

/* Callback/event handle invalid */
/*****************************************************
 * Note: The following code is deprecated.           *
 * Use NPF_E_BAD_CALLBACK_HANDLE instead.            *
 *****************************************************/
```

```
#define NPF_IF_E_BAD_CALLBACK_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+1)

/* Callback function is NULL */
/*****************************************************
 * Note: The following code is deprecated.           *
 * Use NPF_E_BAD_CALLBACK_FUNCTION instead.          *
 *****************************************************/
#define NPF_IF_E_BAD_CALLBACK_FUNCTION ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+2)

/* Invalid parameter */
#define NPF_IF_E_INVALID_PARAM ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+3)

/* Invalid child i/f handle */
#define NPF_IF_E_INVALID_CHILD_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+4)

/* Invalid parent i/f handle */
#define NPF_IF_E_INVALID_PARENT_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+5)

/* Invalid interface handle */
#define NPF_IF_E_INVALID_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+6)

/* Invalid interface attribute */
#define NPF_IF_E_INVALID_ATTRIBUTE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+7)

/* Error – interface not created */
#define NPF_IF_E_NOT_CREATED ((NPF_IfErrorType_t) NPF_INTERFACES_BASE_ERR+8)

/* Array length <= 0 or too big */
#define NPF_IF_E_BAD_ARRAY_LENGTH ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+9)

/* Invalid Interface Type */
#define NPF_IF_E_INVALID_IF_TYPE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+10)

/* Invalid Administrative Status code */
#define NPF_IF_E_INVALID_ADMIN_STATUS ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+11)

/* Parent/child binding not found */
#define NPF_IF_E_NO_SUCH_BINDING ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+12)

/* Parent/child binding is circular */
#define NPF_IF_E_CIRCULAR_BINDING ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+13)

/* Invalid Maximum PDU Size parameter
#define NPF_IF_E_INVALID_MAX_PDU_SIZE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+14)

/* Invalid layer 3 i/f handle */
#define NPF_IF_E_INVALID_L3_HANDLE  ((NPF_IfErrorType_t)
```

```
NPF_INTERFACES_BASE_ERR+15)

/* Interface has no source addr. */
#define NPF_IF_E_NO_SRC_ADDRESS     ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+16)

/* Forwarding is not defined for this interface type */
#define NPF_IF_E_FORWARDING_NOT_DEFINED  ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+17)

/* Insufficient memory to complete the request */
#define NPF_IF_E_NOMEMORY ((NPF_IfErrorType_t) NPF_INTERFACES_BASE_ERR+18)

typedef NPF_uint32_t NPF_IfErrorType_t;
```

3.4 Data Structures for Event Notifications

Events defined here are the "generic" ones: those that can be valid for any interface type. Other type-specific events may be defined in the documents related to particular interface types.

3.4.1 Event Types: NPF_IfEvent_t

Events defined here are the generic ones that may apply regardless of interface type. Additional events may be defined in type-specific Interface Management Implementation agreements; in these cases, they must qualify the event type codes using the interface type code, as described in section 3.1.4, so their event codes will be unique.

```
/*
 *    Event types
 */
#define NPF_IF_EV_UP                1     /* Interface went oper UP */
#define NPF_IF_EV_DOWN              2     /* Interface went oper DOWN */
#define NPF_IF_EV_COUNTER_DISCONTINUITY 3 /* Counter discontinuity occurred*/
#define NPF_IF_EV_CREATED           4     /* Interface was created */
#define NPF_IF_EV_DELETED           5     /* Interface was deleted */
#define NPF_IF_EV_BINDING_CHANGE    6     /* A parent-child binding changed*/
#define NPF_IF_EV_ADDRESS_CHANGE    7     /* L2 or L3 Address changed */
#define NPF_IF_EV_SPEED_CHANGE      8     /* Speed change */
#define NPF_IF_EV_FWD_CHANGE        9     /* Forwarding mode chg */

typedef NPF_uint32_t NPF_IfEvent_t;
```

3.4.2 Event Mask Structure

This structure is used when registering for events. It deviates from NPF Software Conventions, which recommends using a single 32-bit word as an event-enable bit mask, because Interface Management needs a way to allow a sufficient number of bits to be defined for each of an open-ended number of interface types.

The convention here is that there is one event bit mask word for each interface type, and when registering for events, the client can pass an array of masks that includes just the bit mask words it needs for the interface types supported by the handler function. In the array, each mask is accompanied by word containing the Interface Type code for the type of interface it represents. Core events, defined in section 3.4.3, use an ifType code of zero. The event masks can appear in the array in any order, and all types need not be represented. A mask value of zero selects no events for the given interface type. A mask value of all one bits selects all events for the given

interface type. An empty array turns off all interface events notifications, but leaves the handler registration intact and the event registration handle valid.

```
/*
 * Event bit mask specification
 * The client supplies an array of these, one for each
 * interface type, when registering for events on a given
 * set of interfaces.  A "type" code of zero accompanies the
 * mask for Core events.
 */
typedef struct    {
      NPF_IfType_t      ifType;     /* Type designator for this mask */
      NPF_uint32_t      evMask;     /* Event bit mask for this type */
} NPF_IfEvMaskSpec_t;

/*
 * Event bit mask array
 * Passed by the client to the event registration function.
 */
typedef struct    {
      NPF_uint32_t      nMasks;     /* Number of masks in the array */
      NPF_IfEvMaskSpec_t *evMaskArray /* Pointer to array of masks */
} NPF_IfEvMaskArray_t;
```

3.4.3 Core Event Mask Bit Assignments

The following bit assignments for the **`NPF_eventMaskSpec_t`** parameter to the **`NPF_IfEventRegister()`** function are for events defined in the Interface Management Core document only. Additional event mask definitions can be defined in type-specific Interface Management documents.

```
#define NPF_IF_EVMASK_UP            (1<<0)      /* Interface went oper UP */
#define NPF_IF_EVMASK_DOWN          (1<<1)      /* Interface went oper DOWN
*/
#define NPF_IF_EVMASK_COUNTER_DISCONTINUITY (1<<2)    /* Counter
discontinuity occurred*/
#define NPF_IF_EVMASK_CREATED       (1<<3)      /* Interface was created */
#define NPF_IF_EVMASK_DELETED       (1<<4)      /* Interface was deleted */
#define NPF_IF_EVMASK_BINDING_CHANGE (1<<5)     /* A parent-child binding
changed*/
#define NPF_IF_EVMASK_ADDRESS_CHANGE (1<<6)     /* L2 or L3 Address changed
*/
#define NPF_IF_EVMASK_SPEED_CHANGE  (1<<7)      /* Speed change */
#define NPF_IF_EVMASK_FWD_CHANGE    (1<<8)      /* Forwarding mode chg */

#define NPF_IF_EVMASK_ALL                0xFFFFFFFF
```

3.4.4 Event Notification Structure and Array: NPF_IfEventData_t and NPF_IfEventArray_t

The event notification structure contains the type of event, the handle and ID of the interface on which the event occurred, its interface type, and an optional pointer to a structure with

information specific to the type of event. In the case of a binding change event, the interface referred to by the handle, ID and type code in the first part of the structure is the parent interface.

```
/*
 *    Event notification structure and array
 */
typedef struct NPF_IfEventData      {
      NPF_IfEvent_t     eventType;        /* Event type */
      NPF_IfHandle_t    ifHandle;         /* Interface Handle */
      NPF_IfID_t        ifID;             /* Interface ID */
      NPF_IfType_t      ifType;           /* Interface Type */
      union {

      /* **** CAUTION ****
       * EACH MEMBER OF THIS UNION MUST BE THE SAME SIZE,
       * EQUAL TO THE SIZE OF A POINTER VARIABLE.
       * **** CAUTION **** */

      /* For generic functions */

        void *    unused;                 /* Up/down, create/delete events */
        NPF_uint64_t      *speed;         /* new speed in Kbits/second */
        NPF_IfL2L3Addr_Update_t       *L3addrUpdate; /* IP address updates */
        NPF_IfFwdMode_t             *fwdMode;   /* new forwarding mode */
        NPF_IfBind_Update_t         *ifBindUpd; /* new Parent-Child binding*/

      /* For type-specific functions */

      /*
       * The implementer must add lines similar to the above,
       * as needed for the events generated by interface types
       * included in the implementation.
       */

      } u;
} NPF_IfEventData_t;

typedef struct    {
      NPF_uint16_t       n_data;          /* Number of events in array */
      NPF_IfEventData_t *eventData;       /* Array of event notifications */
} NPF_IfEventArray_t;

typedef NPF_uint32_t NPF_IfEventHandlerHandle_t;
```

4 Functions

The Interface management API will provide for setting the interface properties and reading statistics in accordance with the Interface MIB, RFC 2863 [1] and other MIBs (although it makes no attempt to support any MIB fully).

4.1 Completion Callback

4.1.1 Completion Callback Function

Syntax

```
typedef void (*NPF_IfCallbackFunc_t) (
      NPF_IN NPF_userContext_t      userContext,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_IfCallbackData_t  ifCallbackData);
```

Description

The application registers this asynchronous response handling routine to the API implementation. The callback function is implemented by the application, and is registered to the API implementation through the **`NPF_IfRegister()`** function.

This function definition is shared by all Interface Management API Implementation Agreements, including the type-specific ones.

The callback data structure contains an array of responses, so that callbacks for multiple interfaces or ATM UNI Vccs referenced in a single API function call can be aggregated into fewer (perhaps just one) callback function invocations. The application can expect to receive exactly the same number of responses (callback array elements) as the multiplicity of the request, but the responses may be spread over multiple callback function invocations. How the API implementation allocates responses to callback invocations is up to the API implementor.

As an optimization: if the implementation is able to return success indications (**`NPF_NO_ERROR`**) for all responses from a single request in a single invocation of the callback function, and there is no information to return besides the success/failure code: instead of returning an array of responses, the implementation SHALL return a simple code indicating that all requested actions completed without error. See section 3.2.3.

Input Parameters

- **`userContext:`** The context item that was supplied by the application when the completion callback function was registered.
- **`correlator:`** The correlator item that was supplied by the application when the an API function call was made. The correlator is used by the application mainly to distinguish between multiple invocations of the same function.

`ifCallbackData`: A structure containing an array of response information related to the API function call. Contains information that is common among all functions, as well as information specific to a particular function. See **`NPF_IfCallbackData_t`** definition for detailsetails.

`Output Parameters`

None

Return Codes

None

4.1.2 NPF_IfRegister: Completion Callback Registration Function

Syntax

```
NPF_error_t NPF_IfRegister(
      NPF_IN  NPF_userContext_t     userContext,
      NPF_IN  NPF_IfCallbackFunc_t  ifCallbackFunc,
      NPF_OUT NPF_callbackHandle_t *ifCallbackHandle);
```

Description

This function is used by an application to register its completion callback function for receiving asynchronous responses related to API function calls. The application may register multiple callback functions using this function. The callback function is identified by the pair of **`userContext`** and **`ifCallbackFunc`**, and for each individual pair, a unique **`ifCallbackHandle`** will be assigned for future reference. Since the callback function is identified by both **`userContext`** and **`ifCallbackFunc`**, duplicate registration of the same callback function with different **`userContext`** is allowed. Also, the same **`userContext`** can be shared among different callback functions. Duplicate registration of the same **`userContext`** and **`ifCallbackFunc`** pair has no effect, will output a handle that is already assigned to the pair, and will return **`NPF_IF_E_ALREADY_REGISTERED`**.

This function definition is shared by all Interface Management API Implementation Agreements, including the type-specific ones.

Note: **`NPF_IfRegister()`** is a synchronous function and has no completion callback associated with it.

Input Parameters

- **`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. Application can assign any value to the `userContext` and the value is completely opaque to the API implementation.
- **`ifCallbackFunc`**: Pointer to the completion callback function to be registered.

Output Parameters

- **`ifCallbackHandle`**: A unique identifier assigned for the registered `userContext` and `ifCallbackFunc` pair. This handle will be used by the application to specify which callback to be called when invoking asynchronous API functions. It will also be used when de-registering the `userContext` and `ifCallbackFunc` pair.

Return Codes

- **`NPF_NO_ERROR`**: The registration completed successfully.
- **`NPF_IF_E_BAD_CALLBACK_FUNCTION`**: `ifCallbackFunc` is `NULL`.
- **`NPF_IF_E_ALREADY_REGISTERED`**: No new registration was made since the `userContext` and `ifCallbackFunc` pair was already registered**.**
  Note: Whether or not this should be treated as an error is dependent on the application.

4.1.3 NPF_IfDeregister: Completion Callback Deregistration Function

Syntax

```
NPF_error_t NPF_IfDeregister(
      NPF_IN NPF_callbackHandle_t ifCallbackHandle);
```

Description

This function is used by an application to de-register a pair of user context and callback function. After the Deregister function returns, no more function calls can be made using the deregistered callback handle.

This function definition is shared by all Interface Management API Implementation Agreements, including the type-specific ones.

Input Parameters

- **`ifCallbackHandle:`** The unique identifier representing the pair of user context and callback function to be de-registered.

Output Parameters

None

Return Codes

- **`NPF_NO_ERROR`**: The de-registration completed successfully.
- **`NPF_IF_E_BAD_CALLBACK_HANDLE`**: The API implementation does not recognize the callback handle. There is no effect to the registered callback functions.

4.2 Event Notification

4.2.1 Event Handler Function

Syntax

```
typedef void (*NPF_IfEventHandlerFunc_t) (
      NPF_IN NPF_userContext_t  userContext,
      NPF_IN NPF_IfEventArray_t ifEventArray);
```

Description

This handler function is for the application to register an event handling routine to the API implementation. One or more events can be notified to the application through a single invocation of this event handler function. Information on each event is represented in an array in the **`ifEventArray`** structure, where an application can traverse through the array and process each of the events. This event handler function is intended to be implemented by the application, and be registered to the API implementation through **`NPF_IfEventRegister()`** function.

This function definition is shared by all Interface Management API Implementation Agreements, including the type-specific ones.

Note: This function may be called any time after **`NPF_IfEventRegister()`** is called for it.

Input Parameters

- **`userContext`**: The context item that was supplied by the application when the event handler function was registered.
- **`ifEventArray`**: Data structure that contains an array of event information. See **`NPF_IfEventArray_t`** definition for details.

Output Parameters

None

Return Codes

None

4.2.2 NPF_IfEventRegister: Event Handler Registration Function

Syntax

```
NPF_error_t NPF_IfEventRegister(
      NPF_IN  NPF_userContext_t           userContext,
      NPF_IN  NPF_IfEventHandlerFunc_t    ifEventHandlerFunc,
      NPF_IN  NPF_IfEvMaskArray_t               evMaskArray,
      NPF_OUT NPF_IfEventHandlerHandle_t *ifEventHandlerHandle);
```

Description

This function is used by an application to register its event handler function for receiving asynchronous event notifications from this API. Application may register multiple handler functions using this function. The event handler function is identified by the pair of **`userContext`** and **`ifEventHandlerFunc`**, and for each individual pair, a unique **`ifEventHandlerHandle`** will be assigned for future reference. Since the event handler function is identified by both **`userContext`** and **`ifEventHandlerFunc`**, duplicate registration of same event handler function with different **`userContext`** is allowed. Also, same **`userContext`** can be shared among different event handler functions. Duplicate registration of the same **`userContext`** and **`ifEventHandlerFunc`** pair has no effect, and will output a handle that is already assigned to the pair, and will return **`NPF_IF_E_ALREADY_REGISTERED`**.

This function definition is shared by all Interface Management API Implementation Agreements, including the type-specific ones.

Notes: Besides registering a handler function, this call enables events. The handler function could be called at any time following the invocation of **`IfEventRegister()`**. **`NPF_IfEventRegister()`** is a synchronous function and has no completion callback associated with it.

Input Parameters

- **`userContext`**: A context item for uniquely identifying the context of the application registering the event handler function. The exact value will be provided back to the registered event handler function as its first parameter when it is called. Application can assign any value to the userContext and the value is completely opaque to the API implementation**.**
- **`ifEventHandlerFunc`**: Pointer to the event handler function to be registered.
- **`evMaskArray`**: Mask array with a mask word for each requested interface type, and within each mask, a bit set for each event to be enabled on this registration. `See section 3.4.2 for more information.`

Output Parameters

- **`ifEventHandlerHandle`**: A unique identifier assigned for the registered `userContext` and `ifEventHandlerFunc` pair. This handle will be used by the application de-registering the `userContext` and `ifEventHandlerFunc` pair.

Return Codes

- **`NPF_NO_ERROR`**: The registration completed successfully.
- **`NPF_IF_E_BAD_CALLBACK_HANDLE`**: `ifEventHandlerFunc` is `NULL` or not recognized.
- **`NPF_IF_E_ALREADY_REGISTERED`**: No new registration was made since the `userContext` and `ifEventHandlerFunc` pair was already registered.
  Note: Whether or not this should be treated as an error is dependent on the application.

4.2.3 NPF_IfEventDeregister: Event Handler Deregistration Function

Syntax

```
NPF_error_t NPF_IfEventDeregister(
```

```
    NPF_IN NPF_IfEventHandlerHandle_t ifEventHandlerHandle);
```

Description

This function is used by an application to de-register a pair of user context and event handler function.

This function definition is shared by all Interface Management API Implementation Agreements, including the type-specific ones.

Input Parameters

- **`ifEventHandlerHandle`**: The unique identifier representing the pair of user context and event handler function to be de-registered.

Output Parameters

None

Return Codes

- **`NPF_NO_ERROR`**: The de-registration completed successfully.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: The API implementation does not recognize the event handler handle. There is no effect to the registered event handler functions.

4.3 Event Definition Signature

NPF Interfaces can generate the following events:

- **`NPF_IF_UP`** indicates the interface's OperUp status became FALSE
- **`NPF_IF_DOWN`** indicates the interface's OperUp status became TRUE
- **`NPF_IF_COUNTER_DISCONTINUITY`** indicates a discontinuity occurred in one or more of the statistics counters belonging to the interface. This event is intended to help a MIB implementation support **`ifCounterDiscontinuityTime`** (RFC 2863 [1]).

4.4 Order of Operations

There are a few restrictions on the order of operations on interfaces:

1. **`NPF_IfCreate()`** or **`NPF_IfCreateAndSet()`** must precede any other operations on an interface, because those functions assign the **`if_Handle`** value required by all other functions.
2. **`NPF_IfATM_VccSET()`** must precede any other operations on an ATM UNI Vcc.
3. There are no other restrictions, except as may be imposed by a particular implementation.

4.5 Completion Callbacks and Error Returns

Each of the functions defined in section 4.6 can return an immediate error, and each makes asynchronous callbacks. The only error codes eligible for immediate return are those defined in "NPF Software API Conventions Implementation Agreement". They are:

- **`NPF_NO_ERROR`**: This value is returned when a function was successfully invoked.
- **`NPF_E_UNKNOWN`**: An unknown error occurred in the implementation such that there is no error code defined that is more appropriate or informative.
- **`NPF_BAD_CALLBACK_HANDLE`**: A function was invoked with a callback handle that did not correspond to a valid NPF callback handle as returned by a registration function, or a callback handle was registered with a registration function belonging to a different API than the function call where the handle was passed in.
- **`NPF_E_BAD_CALLBACK_FUNCTION`**: A callback registration was invoked with a function pointer parameter that was invalid.

All other error codes must be returned in an asynchronous callback response. They are defined in section 4.6 with the definitions of the functions that return them.

4.6 Interface Management API – Generic Functions

This section will define functions for querying and modifying the interface properties and attributes.

Note: These functions follow a convention permitting multiple interface handles or ATM Vcc addresses to be passed for action in a single function invocation. In each case there is an argument that indicates the size of the array of interface handles or addresses. No limit on the size of such arrays is specified by this agreement; however an implementation MAY impose a size limit of its own choosing. If an application exceeds such limit, the implementation SHALL return the response code **`NPF_IF_E_BAD_ARRAY_LENGTH`** synchronously.

Note: Functions designated as "optional," when not implemented, SHALL return the error code NPF_E_FUNCTION_NOT_SUPPORTED synchronously, per the recommendation of the NP Forum Software Conventions Implementation Agreement.

4.6.1 NPF_IfCreate: Create an Interface

Syntax

```
NPF_error_t  NPF_IfCreate(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_if,
      NPF_IN NPF_IfType_t           if_Type,
      NPF_IN NPF_IfID_t             *ifID);
```

Description

This function creates one or more interfaces of a given type, including "typeless" (type unknown). Interfaces created by this function are in the Administratively Disabled (**`NPF_IF_ADMIN_STATUS_DOWN`**) state by default. The newly created interfaces are all alike, and blank except for type. The callback function will receive as many handles as **`NPF_IfCreate()`** could successfully create, and error codes for the rest. The created interfaces are undifferentiated until you set some attributes in them using **`NPF_IfAttrSet()`** or other functions.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`n_if`**: number of interfaces to create.
- **`if_Type`**: the interface type: **`NPF_IF_TYPE_LAN`**, **`NPF_IF_TYPE_IPv4`**, **`NPF_IF_TYPE_ATM`**, **`NPF_IF_TYPE_POS`**, or **`NPF_IF_TYPE_UNK`**. All interfaces created by one function invocation are of the same type.
- **`ifID`**: a pointer to an array of Interface ID values. The number of elements in the array is given by **`n_if`**. The values must all be different from each other, and none may be the same as the ID of an existing interface.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: operation successful.

- **NPF_E_RESOURCE_EXISTS**: an interface with the same Interface ID value already exists; its handle is returned in the callback, and no new interface is created.
- **NPF_IF_E_INVALID_PARAM**: operation failed, interface not created.

Asynchronous Response

A total of **n_if** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains the new interface handle or a possible error code. The union in the callback response structure is unused.

4.6.2 NPF_IfDelete: Delete an Interface

Syntax

```
NPF_error_t  NPF_IfDelete(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function deletes one or more interfaces. The handle may not be used after this call returns.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces to delete.
- **if_HandleArray**: pointer to an array of handles of the interfaces to be deleted.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_PARAM**: Interface not deleted.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains the handle of the deleted interface, or a possible error code. The union in the callback response structure is unused.

4.6.3 NPF_IfBind: Bind Interfaces

Syntax

```
NPF_error_t  NPF_IfBind(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           nbinds,
      NPF_IN NPF_IfBinding_t       *if_bindArray);
```

Description

This function binds one or more pairs of interfaces in parent-child relationships. Each binding associates two interfaces with each other, one as parent, and one as child. Multiple bindings can be made in a single call. An interface can have multiple parents; it can also have multiple children. Such relationships are indicated by multiple one-to-one binding entries, since a single many-to-one binding entry is not supported. An interface can be at the same time the parent of one and the child of another. An implementation SHOULD return an error if cycles occur (e.g. an interface is the child of one of its own children: "I'm my own grandpa"). An implementation MAY limit how many associations an interface can have, or restrict the depth of the hierarchy.

Bindings have the following characteristics:

- Adding a parent to an interface can mean that a particular protocol can be carried on the link represented by the child.
- Setting a parent Administratively UP or DOWN controls the processing of the protocol represented by the parent; for instance, setting an IPv4 interface down would cause all incoming IPv4 packets received on any of that interface's child interfaces to be discarded.
- Removing a binding does not result in either interface being deleted.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`nbinds`**: number of bindings in the array.
- **`if_bindArray`**: pointer to an array of interface handle parent/child bindings.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: Operation successful.
- **`NPF_IF_E_INVALID_CHILD_HANDLE`**: Child Handle is null or invalid; no binding done.
- **`NPF_IF_E_INVALID_PARENT_HANDLE`**: Parent Handle is null or invalid; no binding done.
- **`NPF_IF_E_INVALID_PARAM`**: Binding failed. No binding done.
- **`NPF_IF_E_CIRCULAR_BINDING`**: An interface would exist more than once in its own parent/child hierarchy. Binding failed; no binding done.

Asynchronous Response

A total of **`n_binds`** asynchronous responses (**`NPF_IfAsyncResponse_t`**) will be passed to the callback function, in one or more invocations. Each response contains the parent interface handle and a possible error code. The particular binding to which the response code pertains is identified in the callback by the two handles: the parent handle is in the usual **`ifHandle`** position, and the child handle is in the union part of the callback structure.

4.6.4 NPF_IfUnBind: Remove Interface Bindings

Syntax

```
NPF_error_t  NPF_IfUnBind(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
```

```
        NPF_IN NPF_uint32_t           nbinds,
        NPF_IN NPF_IfBinding_t       *if_bindArray);
```

Description

This function removes one or more interface parent-child relationships previously set by **NPF_IfBind()** calls.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **nbinds**: number of bindings in the array.
- **if_bindArray**: pointer to an array of interface handle parent/child bindings to be removed.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_NO_SUCH_BINDING**: A specified binding could not be found.

Asynchronous Response

A total of **n_binds** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains the parent interface handle and a possible error code. The particular binding to which the response code pertains is identified in the callback by the two handles: the parent handle is in the usual **ifHandle** position, and the child handle is in the union part of the callback structure.

4.6.5 NPF_IfGenericStatsGet: Read Interface Statistics

Syntax

```
NPF_error_t  NPF_IfGenericStatsGet(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function returns, via a callback, a pointer to a generic interface statistics structure containing the current counter values for one or more indicated interfaces.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces to get statistics for.
- **if_HandleArray**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE:** An **if_Handle** is null or invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle or a possible error code. If the error code indicates success, the union in the callback response structure contains a pointer to the **NPF_IfStatistics_t** structure for that interface.

4.6.6 NPF_IfAttrSet: Set All Interface Attributes

Syntax

```
NPF_error_t  NPF_IfAttrSet(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray,
      NPF_IN NPF_IfGeneric_t       *if_StructArray);
```

Description

This function sets all the attributes of one or more interfaces, from the contents of an array of structures passed by the caller, as defined in **NPF_IfGeneric_t**. Ownership of the structure memory remains with the caller (the API implementation must copy all needed contents before returning). Any single attribute can be set with its own function call; this function is included as a way to set multiple attributes atomically and efficiently. Note: the number of **NPF_IfGeneric_t** structures and the number of interface handles in the two arrays must be the same, equal to the **n_handles** argument. This function sets a *different* set of attributes for each named interface. The Interface Handle value identifies the interface to be modified; the Interface ID value in the **NPF_IfGeneric_t** structure is ignored.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces to set attributes for.
- **if_HandleArray**: pointer to an array of interface handles.
- **if_StructArray**: pointer to a structure or an array of structures containing the new interface attributes.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_ATTRIBUTE**: An attribute (other than those mentioned below) was invalid.

Generic Interface Errors:

- **NPF_IF_E_INVALID_HANDLE**: `if_Handle` is null or invalid.
- **NPF_IF_E_INVALID_IF_TYPE**: Invalid or unsupported interface type code.
- **NPF_IF_E_INVALID_ADMIN_STATUS**: Invalid administrative status code.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.7 NPF_IfCreateAndSet: Create an Interface and Set All of its Attributes

Syntax

```
NPF_error_t  NPF_IfCreateAndSet(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_if,
      NPF_IN NPF_IfGeneric_t        *if_StructArray);
```

Description

This function simultaneously creates and sets all the attributes of one or more interfaces, from the contents of an array of structures passed by the caller (**NPF_IfGeneric_t**). Each interface is created with a *different* set of attributes. Ownership of the structure memory remains with the caller (the API implementation must copy all contents before returning). Each instance of the **NPF_IfGeneric_t** structure must contain a different, nonzero Interface ID value, and none may be the same as that of an existing interface.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_if**: the number of interfaces to set attributes for.
- **if_StructArray**: pointer to an array of structures containing the new interface attributes.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_E_RESOURCE_EXISTS**: an interface with the same Interface ID value already exists; its handle is returned in the callback, and no new interface is created.
- **NPF_IF_E_INVALID_ATTRIBUTE**: An attribute (other than those mentioned below) was invalid.

Generic Interface Errors:

- **NPF_IF_E_INVALID_HANDLE** : **if_Handle** is null or invalid.
- **NPF_IF_E_INVALID_IF_TYPE**: Invalid or unsupported interface type code.
- **NPF_IF_E_INVALID_ADMIN_STATUS**: Invalid administrative status code.

Asynchronous Response

A total of **n_if** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains the new interface handle and a success code or a possible error code if an interface could not be created or any attributes could not be set. Responses are linked to interface attributes in the following way: for each response, the union in the response structure contains the corresponding index of the **if_StructArray** element that contained its attributes. For example, the response for the first array element will include an Interface Handle and an **arrayIndex** value of zero; the response for the tenth array element an **arrayIndex** of 9, and so on.

4.6.8 NPF_IfEnable: Enable an Interface

Syntax

```
NPF_error_t  NPF_IfEnable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function administratively enables one or more interfaces: if the interface is operationally ready, it can now send and receive packets.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces to enable.
- **if_HandleArray**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE:** An **if_Handle** is null or invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.9 NPF_IfDisable: Disable an Interface

Syntax

```
NPF_error_t  NPF_IfDisable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
```

```
            NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function disables one or more interfaces, administratively (but not operationally). Once disabled, it can no longer send or receive packets.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`n_handles`**: the number of interfaces to disable.
- **`if_HandleArray`**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: Operation successful.
- **`NPF_IF_E_INVALID_HANDLE:`** an **`if_Handle`** is null or invalid.

Asynchronous Response

A total of **`n_handles`** asynchronous responses (**`NPF_IfAsyncResponse_t`**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.10 NPF_IfOperStatusGet: Return the Operational Status of an Interface

Syntax

```
NPF_error_t  NPF_IfOperStatusGet(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function returns the operational status of one or more interfaces.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`n_handles`**: the number of interfaces to query.
- **`if_HandleArray`**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: Operation successful.

- **NPF_IF_E_INVALID_HANDLE:** An **if_Handle** is null or invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. If the code indicates success, the union in the callback response structure contains the operational status of the interface.

4.6.11 NPF_IfMaxPDU_SizeSet: Set an Interface's Maximum PDU Size

Syntax

```
NPF_error_t  NPF_IfMaxPDU_SizeSet(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray,
      NPF_IN NPF_uint16_t          *maxPDU_Array);
```

Description

This function sets the Maximum PDU size of one or more interfaces. The **if_HandleArray** and **maxPDU_Array** arrays must both contain the same number of entries, equal to the value of n_handles. The maximum PDU size of each interface is set from a *different* element of the **maxPDU** array.

Note: for an IPv4 or IPv6 interface, maximum PDU size is also known as Maximum Transmission Unit (MTU), meaning the largest IP datagram the interface can accommodate.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces to set the maximum PDU size for.
- **if_HandleArray**: the handle of each interface.
- **maxPDU_Array**: the corresponding maximum PDU size values to be set.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE**: An **if_Handle** is null or invalid, or is not an IPv6 interface.
- **NPF_IF_E_INVALID_MAX_PDU_SIZE**: Maximum PDU size value is invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.12 NPF_IfAttrGet: Read Interface Attributes

Syntax

```
NPF_error_t  NPF_IfAttrGet(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function returns, via a callback, a pointer to a generic interface structure (**`NPF_IfGeneric_t`**) containing the current attributes of one or more indicated interfaces. **This is an optional function.**

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`n_handles`**: the number of interfaces to get attributes for.
- **`if_HandleArray`**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: Operation successful.
- **`NPF_IF_E_INVALID_HANDLE`**: An **`if_Handle`** is null or invalid.

Asynchronous Response

A total of **`n_handles`** asynchronous responses (**`NPF_IfAsyncResponse_t`**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle or a possible error code. If the error code indicates success, the union in the callback response structure contains a pointer to the **`NPF_IfGeneric_t`** structure for that interface.

4.6.13 NPF_IfFwdEnable: Enable Forwarding on One or More Interfaces

Syntax

```
NPF_error_t  NPF_IfFwdEnable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function enables forwarding on one or more interfaces, if a forwarding function is defined for the specific interface type.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.

- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces on which to enable forwarding.
- **if_HandleArray**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE**: An interface handle is null or invalid, or not an IPv4 or IPv4 Tunnel interface.
- **NPF_IF_E_FORWARDING_NOT_DEFINED**: Forwarding is not a defined function for this interface type.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.14 NPF_IfFwdDisable: Disable Forwarding on One or More Interfaces

Syntax

```
NPF_error_t  NPF_IfFwdDisable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function disables forwarding on one or more interfaces. When forwarding is disabled, the interface can still send and receive datagrams as long as the interface is administratively UP and the underlying L2 interface (if any) is operationally and administratively UP. This function has no meaning for interface types on which no forwarding function is defined.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces on which to disable IP forwarding.
- **if_HandleArray**: pointer to an array of IP or IPv4 Tunnel interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE**: An interface handle is null or invalid, or not an IPv4 or IPv6 interface.

- **NPF_IF_E_FORWARDING_NOT_DEFINED**: Forwarding is not a defined function for this interface type.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.15 NPF_IfInternalLoopbackEnable: Enable Internal Loopback on One or More Interfaces

Syntax

```
NPF_error_t  NPF_IfInternalLoopbackEnable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function enables internal loopback on one or more interfaces, if a such a loopback function is implemented for the specific interface type. When enabled, internal loopback causes packets sent from the local system on this interface to be sent back to the local system instead of to their external destinations.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces on which to enable loopback.
- **if_HandleArray**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE**: An interface handle is null or invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.16 NPF_IfInternalLoopbackDisable: Disable Internal Loopback on One or More Interfaces

Syntax

```
NPF_error_t  NPF_IfInternalLoopbackDisable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
```

```
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function turns off internal loopback on one or more interfaces. This function has no meaning for interface types on which no internal loopback function is defined.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`n_handles`**: the number of interfaces on which to disable loopback.
- **`if_HandleArray`**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: Operation successful.
- **`NPF_IF_E_INVALID_HANDLE`**: An interface handle is null or invalid.

Asynchronous Response

A total of **`n_handles`** asynchronous responses (**`NPF_IfAsyncResponse_t`**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.17 NPF_IfExternalLoopbackEnable: Enable External Loopback on One or More Interfaces

Syntax

```
NPF_error_t  NPF_IfExternalLoopbackEnable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function enables external loopback on one or more interfaces, if a such a loopback function is implemented for the specific interface type. When enabled, external loopback causes packets sent from an external system to this interface to be sent back out the external link.

Input Parameters

- **`if_cbHandle`**: the registered callback handle.
- **`if_cbCorrelator`**: the application's context for this call.
- **`if_errorReporting`**: the desired callback.
- **`n_handles`**: the number of interfaces on which to enable loopback.
- **`if_HandleArray`**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE**: An interface handle is null or invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.18 NPF_IfExternalLoopbackDisable: Disable External Loopback on One or More Interfaces

Syntax

```
NPF_error_t  NPF_IfExternalLoopbackDisable(
      NPF_IN NPF_callbackHandle_t   if_cbHandle,
      NPF_IN NPF_correlator_t       if_cbCorrelator,
      NPF_IN NPF_errorReporting_t   if_errorReporting,
      NPF_IN NPF_uint32_t           n_handles,
      NPF_IN NPF_IfHandle_t        *if_HandleArray);
```

Description

This function turns off external loopback on one or more interfaces. This function has no meaning for interface types on which no external loopback function is defined.

Input Parameters

- **if_cbHandle**: the registered callback handle.
- **if_cbCorrelator**: the application's context for this call.
- **if_errorReporting**: the desired callback.
- **n_handles**: the number of interfaces on which to disable loopback.
- **if_HandleArray**: pointer to an array of interface handles.

Output Parameters

None

Asynchronous Error Codes

- **NPF_NO_ERROR**: Operation successful.
- **NPF_IF_E_INVALID_HANDLE**: An interface handle is null or invalid.

Asynchronous Response

A total of **n_handles** asynchronous responses (**NPF_IfAsyncResponse_t**) will be passed to the callback function, in one or more invocations. Each response contains an interface handle and a success code or a possible error code for that interface. The union in the callback response structure is unused.

4.6.19 NPF_IfHandleGet: Return the Handle Value For a Given Interface

Syntax

```
NPF_error_t NPF_IfHandleGet (
```

```
        NPF_IN NPF_callbackHandle_t         if_cbHandle,
        NPF_IN NPF_correlator_t             if_cbCorrelator,
        NPF_IN NPF_errorReporting_t         if_errorReporting,
        NPF_IN NPF_uint32_t                 n_if,
        NPF_IN NPF_IfID_t                   *ifIDArray);
```

Description

This function returns the handle value for one or more interfaces, given their Interface ID values. This is an optional function.

Input Parameters

- **`cbHandle`**: the registered callback handle.
- **`cbCorrelator`**: the application's context for this call.
- **`errorReporting`**: the desired level of feedback
- **`n_if`**: number of interface IDs in the array
- **`ifIDArray:`** pointer to an array of interface IDs.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: The operation was successful.
- **`NPF_IF_E_NOMEMORY`**: The system was unable to allocate sufficient memory to complete this operation.
- **`NPF_IF_E_INVALID_PARAM`**: There is no handle associated with the given interface ID.

Asynchronous response

A callback of type **`NPF_IF_HANDLE_GET`** is generated in response to this function call. For each intrface ID given in the request, the **`NPF_IfAsyncResponse_t`** structure contains the error code (if any), the given ID value, and the returned handle. The union within the **`NPF_IfAsyncResponse_t`** structure is not used.

4.6.20 NPF_IfHandleGetAll: Return the Handles of All Interfaces

Syntax

```
NPF_error_t NPF_IfIdentity_GetAll (
        NPF_IN NPF_callbackHandle_t         if_cbHandle,
        NPF_IN NPF_correlator_t             if_cbCorrelator,
        NPF_IN NPF_errorReporting_t         if_errorReporting);
```

Description

This function returns an array containing the IDs and Interface Handles of all existing interfaces. This is an optional function.

Input Parameters

- **`cbHandle`**: the registered callback handle.
- **`cbCorrelator`**: the application's context for this call.
- **`errorReporting`**: the desired level of feedback.

Output Parameters

None

Asynchronous Error Codes

- **`NPF_NO_ERROR`**: The operation was successful.
- **`NPF_IF_E_NOMEMORY`**: The system was unable to allocate sufficient memory to complete this operation.

Asynchronous response

A callback of type **`NPF_IF_HANDLE_GET_ALL`** is generated in response to this function call. The union within the **`NPF_IfAsyncResponse_t`** structure points to an **`NPF_IfIdentityArray_t`** structure, which in turn points to an array of **`NPF_Identity_t`** structures containing the Interface IDs and handles of all existing interfaces.

5 References

1 McCloughrie, K., Kastenholtz, F., “The Interfaces Group MIB”, Internet Engineering Task Force RFC 2863, June 2000.

2 NP Forum – Software API Conventions Implementation Agreement Revision 2.0.

6 API Capabilities

This section defines the capabilities of the Interface Management API.

It summarizes the defined APIs and Events and defines the mandatory and optional features.

6.1 Optional support of specific types

The support of any specific type of interface is optional in an implementation. An implementation MAY support exclusively one type of interface, and still claim compliance to the NP Forum Interface Management API.

6.2 API Functions

Function Name	Required?
NPF_IfRegister()	Yes
NPF_IfDeregister()	Yes
NPF_IfEventRegister()	Yes
NPF_IfEventDeregister()	Yes
NPF_IfCreate()	Yes
NPF_IfDelete()	Yes
NPF_IfBind()	Yes
NPF_IfUnBind()	Yes
NPF_IfGenericStatsGet()	Yes
NPF_IfAttrSet()	Yes
NPF_IfCreateAndSet()	Yes
NPF_IfEnable()	Yes
NPF_IfDisable()	Yes
NPF_IfOperStatusGet()	Yes
NPF_IfMaxPDU_SizeSet()	Yes
NPF_IfAttrGet	No
NPF_IfFwdEnable()	Only if interface type supports forwarding
NPF_IfwdDisable()	Only if interface type supports forwarding
NPF_IfInternalLoopbackEnable	Only if interface supports internal loopback
NPF_IfInternalLoopbackDisable	Only if interface supports internal loopback
NPF_IfExternalLoopbackEnable	Only if interface supports external loopback
NPF_IfExternalLoopbackDisable	Only if interface supports external loopback
NPF_IfHandleGet()	No
NPF_IfHandleGetAll()	No

6.3 API Events

Event Name	Required?
NPF_IF_UP	Yes
NPF_IF_DOWN	Yes
NPF_IF_COUNTER_DISCONTINUITY	Yes
NPF_IF_EV_CREATED	Yes
NPF_IF_EV_DELETED	Yes
NPF_IF_EV_BINDING_CHANGE	Yes
NPF_IF_EV_ADDRESS_CHANGE	Yes
NPF_IF_EV_SPEED_CHANGE	Yes
NPF_IF_EV_FWD_CHANGE	Yes

APPENDIX A CHANGES FROM REVISION 2.0

The major change in Revision 3.0 of the Interface Management API was to split the Revision 2.0 document into several documents: an Interface Management Core document that defined all the generic parts of the Interface Management API, and for each particular interface type or related group of interface types, a type-specific document that adds API definitions specific to that type or group.

The original document was split in this way to accomplish two things:

1. to create some independence of the documents, so that changes specific to a single interface type, which we found occur very frequently, need not result in constant revisions to a single document; and
2. to make it easier for vendors and customers to understand what is required when an implementation claims support for specific interface types.

Interface Management is still considered to be a single API, although its definitions are now spread across multiple documents. No type-specific document stands alone; they all depend on definitions found in the Core document:

- **`NPF_IfGeneric_t`** structure
- Callback handler and registration functions
- Event handler and registration functions
- Inteface create, delete, query, and other generic function definitions.

Splitting the document could not be done in a completely backward-compatible way. This appendix lists the changes that will require work by implementers to migrate from version 2.0 to version 3.0 of the Implementation Agreement. The changes are as follows:

- Modular header files:
  The monolithic header file of Revision 2.0 was split into **`npf_if_core.h`**, **`npf_if_lan.h`**, **`npf_if_ipv4.h`**, **`npf_if_ipv6.h`**, and **`npf_if_pos.h`**. API implementations should supply the core header file and type-specific header files for all supported types; clients need only include **`npf_if_core.h`** and any additional header files needed for the types they use.
- Embedded structures changed to pointers:
  The **`NPFf_IfGeneric_t`**, **`npf_IfAsyncResponse_t`**, and **`npf_IfEventData_t`** structures each contained a union of type-specific structures. These embedded structures have been changed to pointers, so that new types could be defined without changing the size of the parent structure.
- Common variables with type-specific value assignments:
  `NPF_IfType_t`, **`NPF_IfCallbackType_t`**, and **`NPF_IfEvent_t`** were changed from **`enum`** data types to **`NPF_uint32_t`**, with values assigned using **`#define`** statements. (This allows the code assignments to be spread across the Interface Management document set, and new codes to be added later without having to touch the typedef statement for the variable.) **`NPF_IfErrorType_t`** was already defined in this way, but now its value assignments are scattered as well. The code values for all these variables are no longer squential integers; values assigned in the Core document start from 1 and go sequentially. Values defined in type-specific document start from 1 plus an offset of the interface type code left shifted by

either 8 or 16 bits, depending on the variable. Type-specific error codes now are assigned in the range from 0x10101 through 0x1ffff, so as to avoid conflict with codes assigned by other APIs.

- Replacement of some type-specific functions with generic ones:
  – **`NPF_IfIPv4MTU_Set()`** and **`NPF_IfIPv6MTU_Set()`** functions have been replaced by a generic **`NPF_IfMaxPDU_SizeSet()`**. There is now a generic Max PDU Size attribute that means the same as MTU, and replaces it for IP interfaces.
  – The IP forwarding enable/disable functions, **`NPF_IfIPv[4|6]_FwdEnable()`** and **`NPF_IfIPv[4|6]_FwdDisable()`**, have been replaced by **`NPF_IfFwdEnable()`** and **`NPF_IfFwdDisable()`** generic functions, which can be use with any interface type that supports forwarding operations (such as LAN or ATM as well as L3 types).

APPENDIX B HEADER FILE: NPF_IF_CORE.H

```
/*
 * This header file defines typedefs, constants, and functions
 * that apply to the NPF Core Interface Management API.
 */
#ifndef __NPF_IF_CORE_H__
#define __NPF_IF_CORE_H__

#ifdef __cplusplus
extern "C"  {
#endif


/*
 * Interface Management depends on some types that are defined
 * in other header files because they are shared by other APIs.
 */
#ifndef NPF_MAC_Address_t     /* Should be defined in a common .h file */
typedef NPF_uchar8_t NPF_MAC_Address_t[6];
#endif

#ifndef NPF_IPv4Address_t     /* Should be defined in a common .h file */
typedef NPF_uint32_t NPF_IPv4Address_t;
#endif

#ifndef NPF_IPv4Prefix_t      /* Should be defined in a common .h file */
typedef struct NPF_IPv4Prefix {
        NPF_IPv4Address_t  IPv4Addr;    /* IPv4 address */
        NPF_uint8_t     IPv4NetPlen;    /* Prefix length in bits (1-32) */
} NPF_IPv4Prefix_t;
#endif

#ifndef NPF_IPv6Address_t     /* Should be defined in a common .h file */
typedef struct  {
        union   {
          NPF_uchar8_t  b[16];
          NPF_uint32_t  w[4];
        } u;
} NPF_IPv6Address_t;
#endif


#ifndef NPF_IPv6Prefix_t      /* Should be defined in a common .h file */
/*
 * IPv6 address prefix structure
 */
typedef struct NPF_IPv6Prefix {
        NPF_IPv6Address_t IPv6Addr;     /* IPv6 address */
        NPF_uint8_t     IPv6Plen;       /* Prefix length in bits (1-128) */
} NPF_IPv6Prefix_t;
#endif

#ifndef NPF_IfHandle_t        /* Should be defined in a common .h file */
```

```
/*
 * Interface handle
 */
typedef NPF_uint32_t    NPF_IfHandle_t;
#endif


/*
 * Interface Management Definitions
 */
typedef NPF_uint32_t    NPF_IfID_t;       /* Interface Identifier */

typedef NPF_uint32_t NPF_IfType_t;
#define NPF_IF_TYPE_UNK 1                 /* Interface type unknown */

/*
 *    Structure to relate two interfaces
 */
typedef struct {
      NPF_IfHandle_t    parent;           /* Parent interface handle */
      NPF_IfHandle_t    child;            /* Child interface handle */
} NPF_IfBinding_t;

/*
 *    Statistics
 */
typedef struct {
      NPF_uint64_t      bytesRx;    /* Receive Bytes */
      NPF_uint64_t      ucPackRx;   /* Receive Unicast Packets */
      NPF_uint64_t      mcPackRx;   /* Receive Multicast Packets */
      NPF_uint64_t      bcPackRx;   /* Receive Broadcast packets */
      NPF_uint64_t      dropRx;           /* Receive packets dropped */
      NPF_uint64_t      errorRx;    /* Receive errors */
      NPF_uint64_t      protoRx;    /* Receive unknown protocol */
      NPF_uint64_t      bytesTx;    /* Transmit bytes */
      NPF_uint64_t      ucPackTx;   /* Transmit Unicast Packets */
      NPF_uint64_t      mcPackTx;   /* Transmit Multicast Packets */
      NPF_uint64_t      bcPackTx;   /* Transmit Broadcast Packets */
      NPF_uint64_t      dropTx;           /* Transmit dropped packets */
      NPF_uint64_t      errorTx;    /* Transmit errors */
} NPF_IfStatistics_t;

/*
 *    Operational Status code
 */
typedef enum      {
      NPF_IF_OPER_STATUS_UP = 1,          /* Operationally UP */
      NPF_IF_OPER_STATUS_DOWN = 2,        /* Operationally DOWN */
      NPF_IF_OPER_STATUS_TESTING = 3,     /* Testing status */
      NPF_IF_OPER_STATUS_UNKNOWN = 4,     /* Status unknown */
      NPF_IF_OPER_STATUS_DORMANT = 5,     /* Dormant status */
      NPF_IF_OPER_STATUS_NOT_PRESENT = 6, /* Interface not present */
      NPF_IF_OPER_STATUS_LOWER_LAYER_DOWN = 7 /* Parent I/F down */
} NPF_IfOperStatus_t;

/*
 *    Administrative Status code
```

```
 */
typedef enum      {
      NPF_IF_ADMIN_STATUS_UP = 1,         /* Administratively UP */
      NPF_IF_ADMIN_STATUS_DOWN = 2,       /* Administratively DOWN */
      NPF_IF_ADMIN_STATUS_TESTING = 3     /* Testing status */
} NPF_IfAdminStatus_t;


/*
 *    Forwarding mode code
 */
typedef enum      {
      NPF_IF_FORWARDING_ENABLE = 1, /* Enable Forwarding */
      NPF_IF_FORWARDING_DISABLE = 2 /* Disable Forwarding */
} NPF_IfFwdMode_t;

/*
 * Internal and External Loopback Modes
 */
typedef     enum  {
      NPF_IF_INTERNAL_LOOPBACK_ENABLE     = 1,  /* Enable loopback */
      NPF_IF_INTERNAL_LOOPBACK_DISABLE    = 2,  /* Disable loopback */
} NPF_IfInternalLoopbackMode_t;

typedef     enum  {
      NPF_IF_EXTERNAL_LOOPBACK_ENABLE     = 1,  /* Enable loopback */
      NPF_IF_EXTERNAL_LOOPBACK_DISABLE    = 2,  /* Disable loopback */
} NPF_IfExternalLoopbackMode_t;

/*
 *    Interface Identity (ID and Handle)
 */
typedef struct {
      NPF_IfID_t     ifID;
      NPF_IfHandle_t ifHandle;
} NPF_IfIdentity_t;

/*
 *    Interface Identity Array
 */
typedef struct {
      NPF_uint32_t nCount;
      NPF_IfIdentity_t *ifIdentityArray;
} NPF_IfIdentityArray_t;

/*
 *    The Interface structure:
 */

/*
 * The implementer adds lines like the following, depending
 * on the interface types supported.  In this example, we have
 * support for LAN and IPv4 interface types.
 */
typedef struct NPF_IfLAN      NPF_IfLAN_t;
typedef struct NPF_IfIPv4     NPF_IfIPv4_t;

typedef struct  {
```

```
      NPF_IfID_t          ifID;           /* Interface ID */
      NPF_IfType_t        type;           /* Logical interface type */
      NPF_uint64_t        speed;          /* Speed in Kbits/second */
      NPF_uint32_t        maxPDU;         /* Max Protocol Data Unit Size */
      NPF_IfOperStatus_t operStatus;      /* Operational Status (read only)*/
      NPF_IfAdminStatus_t adminStatus;    /* Administrative up/down */
      NPF_IfFwdMode_t   fwdMode;          /* Forwarding Mode */
      NPF_IfInternalLoopbackMode_t  intLoop;    /* Internal loopback */
      NPF_IfExternalLoopbackMode_t  extLoop;    /* External loopback */
      NPF_uint32_t        nChildren;      /* Number of child interfaces */
      NPF_uint32_t        *childIDs;      /* Array of child interface IDs */
      NPF_uint32_t        nParents; /* Number of parent interfaces */
      NPF_uint32_t        *parentIDs;     /* Array of parent i/f IDs */
      union {       /* Type specific attributes (by if_type code) */

      /* **** CAUTION ****
       * ONLY POINTERS TO STRUCTURES MAY BE USED IN THIS UNION.
       * **** CAUTION **** */

      /* The implementer adds lines like the following,
       * depending on the interface types supported. In
       * this example, we have support for LAN and IPv4
       * interface types.
       */
        NPF_IfLAN_t *LAN_Attr;      /* LAN interface attributes */
        NPF_IfIPv4_t *IPv4_Attr;    /* IPv4 Interface attributes */
      } u;
} NPF_IfGeneric_t;

/*
 * Action for address changes
 */
typedef enum     {
     IF_ADDR_ADD     = 0,
     IF_ADDR_DELETE  = 1,
     IF_ADDR_MODIFY   = 2
} NPF_IfAddrUpdate_Type_t;

/*
 *    L2/L3 Address type
 */
typedef enum     {
  IF_IPV4_ADDR   = 1,  /* modify primary IPv4 address */
  IF_IPV4_UCADDR = 2,  /* Add, delete IPv4 unicast addresses */
  IF_IPV4_MCADDR = 3,  /* Add, delete Multicast addresses */
  IF_IPV6_ADDR = 4,     /* Add, delete IPv6 addresses */
  IF_MAC_ADDR = 5       /* Add, delete MAC addresses */
} NPF_IfL2L3Addr_Type_t;


/*
 *    L2/L3 Address Changes
 */
typedef struct
{
   NPF_IfAddrUpdate_Type_t   addrChangeType; /* add, delete or modify */
   NPF_IfL2L3Addr_Type_t       addrType;     /* Ipv4, Ipv6, MAC addr, etc. */
```

```
   NPF_uint32_t          nAddrs
   union {
      NPF_IPv4Prefix_t   *if_IPv4AddrArray;
      NPF_IPv6Prefix_t   *if_IPv6AddrArray;
      NPF_IfMacAddress_t *macAddrArray;
   } newAddr;
} NPF_IfL2L3Addr_Update_t;


/*
 * Completion Callback Types
 */
typedef NPF_uint32_t NPF_IfCallbackType_t;
#define NPF_IF_CREATE              1
#define NPF_IF_DELETE              2
#define NPF_IF_BIND                3
#define NPF_IF_UN_BIND             4
#define NPF_IF_STATS_GET           5
#define NPF_IF_ATTR_SET            6
#define NPF_IF_CREATE_AND_SET      7
#define NPF_IF_ENABLE              8
#define NPF_IF_DISABLE             9
#define NPF_IF_OPER_STATUS_GET     10
#define NPF_IF_MAX_PDU_SIZE_SET    11
#define NPF_IF_ATTR_GET            12
#define NPF_IF_FWD_ENABLE          13
#define NPF_IF_FWD_DISABLE         14
#define NPF_IF_INTERNAL_LOOPBACK_ENABLE   15
#define NPF_IF_INTERNAL_LOOPBACK_DISABLE  16
#define NPF_IF_EXTERNAL_LOOPBACK_ENABLE   17
#define NPF_IF_EXTERNAL_LOOPBACK_DISABLE  18
#define NPF_IF_HANDLE_GET          19
#define NPF_IF_HANDLE_GET_ALL      20


typedef NPF_uint32_t NPF_IfErrorType_t;

/*
 *    An asynchronous response contains an interface handle,
 *    a error or success code, and in some cases a function-
 *    specific structure embedded in a union.  One or more of
 *    these is passed to the callback function as an array
 *    within the NPF_IfCallbackData_t structure (below).
 */

typedef struct NPF_IfATM_VccStats NPF_IfATM_VccStats_t; /* Forward reference
*/

typedef struct    {                /* Asynchronous Response Structure */
    NPF_IfHandle_t      ifHandle;   /* Interface handle for this response */
    NPF_IfID_t          ifID;       /* Interface ID */
    NPF_IfErrorType_t error;  /* Error code for this response */
    union { /* Function-specific response information:      */

      /* **** CAUTION ****
       * EACH MEMBER OF THIS UNION MUST BE THE SAME SIZE,
```

```
         * EQUAL TO THE SIZE OF A POINTER VARIABLE.
         * **** CAUTION **** */

        /* For generic functions */
        NPF_uint32_t      unused;           /* Default */
        NPF_uint32_t      arrayIndex; /* NPF_IfCreateAndSet index */
        NPF_IfStatistics_t *ifStats;  /* NPF_IfGenericStatsGet() */
        NPF_IfOperStatus_t operStat;  /* NPF_IfOperStatusGet() */
        NPF_IfHandle_t    child;            /* NPF_IfBind(), handle=parent*/
        NPF_IfGeneric_t   *attrs;           /* NPF_IfAttrGet() */
        NPF_IfIdentifyArray_t *idArray      /* NPF_IfHandleGetAll() */

        /* For type-specific functions */

        /*
         * The implementer must add lines like the following,
         * as needed for the responses of functions supporting
         * the interface types included in the implementation.
         * In this example we have support for LAN and IPv4
         * interface types.  Each member must be the same
         * size: the size of a pointer variable.
         */

        NPF_MAC_Address_t *MACaddr;   /* NPF_IfLAN_SrcAddrGet() */
        NPF_IPv4Prefix_t  *v4prefix;  /* NPF_IPv4UC_AddrAdd(), */
                                    /* Set(),Delete() */
        NPF_IPv4Address_t *v4addr;    /* NPF_IPv4McastAddrAdd(), */
                                    /* Set() */
    } u;

} NPF_IfAsyncResponse_t;

/*
 *    The callback function receives the following structure containing
 *    one or more asynchronous responses from a single function call.
 *    There are several possibilities:
 *    1. The called function does a single request
 *       - n_resp = 1, and the resp array has just one element.
 *       - allOK = TRUE if the request completed without error
 *         and the only return value is the response code.
 *       - if allOK = FALSE, the "resp" structure has the error code.
 *    2. the called function supports an array of requests
 *       a. All completed successfully, at the same time, and the
 *          only returned value is the response code:
 *          - allOK = TRUE, n_resp = 0.
 *       b. Some completed, but not all, or there are values besides
 *          the response code to return:
 *          - allOK = FALSE, n_resp = the number completed
 *          - the "resp" array will contain one element for
 *            each completed request, with the error code
 *            in the NPF_IfAsyncResponse_t structure, along
 *            with any other information needed to identify
 *            which request element the response belongs to.
 *          - 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.
 */
typedef struct {
      NPF_IfCallbackType_t    type; /* Which function was called? */
      NPF_boolean_t           allOK;      /* TRUE if all completed OK */
      NPF_uint32_t            n_resp;     /* Number of responses in array */
      NPF_IfAsyncResponse_t  *resp; /* Pointer to response structures*/
} NPF_IfCallbackData_t;

/*
 *    Error codes */

/* Callback/event reg. error */
#define NPF_IF_E_ALREADY_REGISTERED ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR)

/* Callback/event handle invalid */
#define NPF_IF_E_BAD_CALLBACK_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+1)

/* Callback function is NULL */
#define NPF_IF_E_BAD_CALLBACK_FUNCTION ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+2)

/* Invalid parameter */
#define NPF_IF_E_INVALID_PARAM ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+3)

/* Invalid child i/f handle */
#define NPF_IF_E_INVALID_CHILD_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+4)

/* Invalid parent i/f handle */
#define NPF_IF_E_INVALID_PARENT_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+5)

/* Invalid interface handle */
#define NPF_IF_E_INVALID_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+6)

/* Invalid interface attribute */
#define NPF_IF_E_INVALID_ATTRIBUTE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+11)

/* Error ? interface not created */
#define NPF_IF_E_NOT_CREATED ((NPF_IfErrorType_t) NPF_INTERFACES_BASE_ERR+12)

/* Invalid layer 3 i/f handle */
#define NPF_IF_E_INVALID_L3_HANDLE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+16)

/* Array length <= 0 or too big */
#define NPF_IF_E_BAD_ARRAY_LENGTH ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+17)

/* Interface has no source addr. */
```

```
#define NPF_IF_E_NO_SRC_ADDRESS ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+18)

/* Invalid Interface Type */
#define NPF_IF_E_INVALID_IF_TYPE ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+21)

/* Invalid Port number */
#define NPF_IF_E_INVALID_PORT_NUMBER ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+22)

/* Invalid Administrative Status code */
#define NPF_IF_E_INVALID_ADMIN_STATUS ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+22)

/* Parent/child binding not found */
#define NPF_IF_E_NO_SUCH_BINDING ((NPF_IfErrorType_t)
NPF_INTERFACES_BASE_ERR+30)

/*
 * Event bit mask specification
 * The client supplies an array of these, one for each
 * interface type, when registering for events on a given
 * set of interfaces.  A type code of zero accompanies the
 * mask for Core events.
 */
typedef struct    {
      NPF_IfType_t      ifType;     /* Type designator for this mask */
      NPF_uint32_t      evMask;     /* Event bit mask for this type */
} NPF_IfEvMaskSpec_t;

/*
 * Event bit mask array
 * Passed by the client to the event registration function.
 */
typedef struct    {
      NPF_uint32_t      nMasks;     /* Number of masks in the array */
      NPF_IfEvMaskSpec_t *evMaskArray /* Pointer to array of masks */
} NPF_IfEvMaskArray_t;

#define NPF_IF_EVMASK_IF_UP         (1<<0)      /* Interface went oper UP */
#define NPF_IF_EVMASK_IF_DOWN       (1<<1)      /* Interface went oper DOWN
*/
#define NPF_IF_EVMASK_COUNTER_DISCONTINUITY (1<<2)    /* Counter
discontinuity occurred*/
#define NPF_IF_EVMASK_CREATED       (1<<3)      /* Interface was created */
#define NPF_IF_EVMASK_DELETED       (1<<4)      /* Interface was deleted */
#define NPF_IF_EVMASK_BINDING_CHANGE (1<<5)     /* A parent-child binding
changed*/
#define NPF_IF_EVMASK_ADDRESS_CHANGE (1<<6)     /* L2 or L3 Address changed
*/
#define NPF_IF_EVMASK_SPEED_CHANGE  (1<<7)      /* Speed change */
#define NPF_IF_EVMASK_FWD_CHANGE    (1<<8)      /* Forwarding mode chg */

#define NPF_IF_EVMASK_ALL                       0xFFFFFFFF

/*
```

```
 *    Core Event types
 */
typedef NPF_uint32_t NPF_IfEvent_t;
#define NPF_IF_UP 1           /* Interface went oper UP */
#define NPF_IF_DOWN 2         /* Interface went oper DOWN */
#define NPF_IF_COUNTER_DISCONTINUITY 3 /* Counter discontinuity occurred*/
#define NPF_IF_EV_CREATED           4     /* Interface was created */
#define NPF_IF_EV_DELETED           5     /* Interface was deleted */
#define NPF_IF_EV_BINDING_CHANGE    6     /* A parent-child binding changed*/
#define NPF_IF_EV_ADDRESS_CHANGE    7     /* L2 or L3 Address changed */
#define NPF_IF_EV_SPEED_CHANGE      8     /* Speed change */
#define NPF_IF_EV_FWD_CHANGE        9     /* Forwarding mode chg */

/*
 *    Event notification structure and array
 */
typedef struct NPF_IfEventData      {
      NPF_IfEvent_t     eventType;        /* Event type */
      NPF_IfHandle_t    ifHandle;         /* Interface Handle */
      NPF_IfID_t        ifID;             /* Interface ID */
      NPF_IfType_t      ifType;           /* Interface Type */
      union  {

      /* **** CAUTION ****
       * EACH MEMBER OF THIS UNION MUST BE THE SAME SIZE,
       * EQUAL TO THE SIZE OF A POINTER VARIABLE.
       * **** CAUTION **** */

      /* For generic functions */

        void *    unused;                 /* Up/down, create/delete events */
        NPF_uint64_t      *speed;         /* new speed in Kbits/second */
        NPF_IfL2L3Addr_Update_t       *L3addrUpdate; /* IP address updates */
        NPF_IfFwdMode_t             *fwdMode;   /* new forwarding mode */
        NPF_IfBind_Update_t         *ifBindUpd; /* new Parent-Child binding*/

      /* For type-specific functions */

      /*
       * The implementer must add lines similar to the above,
       * as needed for the events generated by interface types
       * included in the implementation.
       */

      } u;
} NPF_IfEventData_t;



typedef struct    {
      NPF_uint16_t       n_data;          /* Number of events in array */
      NPF_IfEventData_t *eventData;       /* Array of event notifications */
} NPF_IfEventArray_t;

typedef     NPF_uint32_t NPF_IfEventHandlerHandle_t;

typedef void (*NPF_IfCallbackFunc_t) (
```

```
    NPF_IN NPF_userContext_t  userContext,
    NPF_IN NPF_correlator_t   correlator,
    NPF_IN NPF_IfCallbackData_t  ifCallbackData);

NPF_error_t NPF_IfRegister(
    NPF_IN  NPF_userContext_t     userContext,
    NPF_IN  NPF_IfCallbackFunc_t  ifCallbackFunc,
    NPF_OUT NPF_callbackHandle_t *ifCallbackHandle);

NPF_error_t NPF_IfDeregister(
    NPF_IN NPF_callbackHandle_t ifCallbackHandle);

typedef void (*NPF_IfEventHandlerFunc_t) (
    NPF_IN NPF_userContext_t  userContext,
    NPF_IN NPF_IfEventArray_t ifEventArray);

NPF_error_t NPF_IfEventRegister(
      NPF_IN  NPF_userContext_t           userContext,
      NPF_IN  NPF_IfEventHandlerFunc_t    ifEventHandlerFunc,
      NPF_IN  NPF_IfEvMaskArray_t         evMaskArray,
      NPF_OUT NPF_IfEventHandlerHandle_t *ifEventHandlerHandle);



NPF_error_t NPF_IfEventDeregister(
    NPF_IN NPF_IfEventHandlerHandle_t ifEventHandlerHandle);

NPF_error_t  NPF_IfCreate(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_if,
    NPF_IN NPF_IfType_t             if_Type,
    NPF_IN NPF_IfID_t               *ifID);

NPF_error_t  NPF_IfDelete(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfBind(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             nbinds,
    NPF_IN NPF_IfBinding_t         *if_bindArray);

NPF_error_t  NPF_IfUnBind(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             nbinds,
    NPF_IN NPF_IfBinding_t         *if_bindArray);

NPF_error_t  NPF_IfGenericStatsGet(
```

```
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfAttrSet(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray,
    NPF_IN NPF_IfGeneric_t         *if_StructArray);

NPF_error_t  NPF_IfCreateAndSet(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_if,
    NPF_IN NPF_IfGeneric_t          *if_StructArray);

NPF_error_t  NPF_IfEnable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfDisable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfOperStatusGet(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfMaxPDU_SizeSet(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray,
    NPF_IN NPF_uint16_t            *maxPDU_Array);

NPF_error_t  NPF_IfAttrGet(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);
```

```
NPF_error_t  NPF_IfFwdEnable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfFwdDisable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfInternalLoopbackEnable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfInternalLoopbackDisable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfExternalLoopbackEnable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t  NPF_IfExternalLoopbackDisable(
    NPF_IN NPF_callbackHandle_t     if_cbHandle,
    NPF_IN NPF_correlator_t   if_cbCorrelator,
    NPF_IN NPF_errorReporting_t     if_errorReporting,
    NPF_IN NPF_uint32_t             n_handles,
    NPF_IN NPF_IfHandle_t          *if_HandleArray);

NPF_error_t NPF_IfHandleGet (
    NPF_IN NPF_callbackHandle_t           if_cbHandle,
    NPF_IN NPF_correlator_t               if_cbCorrelator,
    NPF_IN NPF_errorReporting_t           if_errorReporting,
    NPF_IN NPF_uint32_t                 n_if,
    NPF_IN NPF_IfID_t                   *ifIDArray);

NPF_error_t NPF_IfIdentity_GetAll (
    NPF_IN NPF_callbackHandle_t           if_cbHandle,
    NPF_IN NPF_correlator_t               if_cbCorrelator,
    NPF_IN NPF_errorReporting_t           if_errorReporting);

#ifdef __cplusplus
}
#endif
```

```
#endif
```

APPENDIX C ACKNOWLEDGEMENTS

Working Group Chair:

Alex Conta, Transwitch, aconta@txc.com

Task Group Chair:

Alex Conta, Transwitch, aconta@txc.com

Task Group Editor:

John Renwick, Agere Systems, jrenwick@agere.com

The following individuals are acknowledged for their participation to IM API TG teleconferences, plenary meetings, mailing list, and/or for their NPF contributions used for the development of this Implementation Agreement. This list may not be all-inclusive since only names supplied by member companies for inclusion here will be listed. The NPF wishes to thank all active participants to this Implementation Agreement, whether listed here or not.

The list is in alphabetical order of last names: