Platform Namespace API

Reference Guide

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Figures

Tables

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Shailesh Suman
2.1	Updated for Release 2.1	December 2003	Shailesh Suman
2.0	Updated for Release 2.0	August 2003	Shailesh Suman

Part 1: Overview

1 Overview

Network elements such as switches and routers can be classified into three logical operational components: Control plane, Forwarding plane, and Management plane.

The control plane controls and configures the forwarding plane. The control plane executes different signaling or routing protocols and provides all the routing information to the forwarding plane.

The forwarding plane manipulates network traffic and makes decisions based on this information. The forwarding plane performs operations on packets such as forwarding, classification, filtering, and so on.

An orthogonal management plane manages the control and forwarding planes.

For example, the control plane in a router executes routing protocols, the forwarding plane performs hardware-based switching, and the management plane starts or stops routing process or performs logging.

The introduction of standardized APIs within the above-mentioned planes can help system vendors, OEMs, and end users of these network elements to mix and match components available from different vendors to achieve a device of their choice. The Network Processing Forum (NPF) API is designed for this purpose, as it presents a flexible and well-known programming interface to the control plane applications. It makes the existence of multiple forwarding planes, as well as vendor-specific details, transparent to control plane applications. Furthermore, the hardware properties and nature of interconnect used between the control and the forwarding planes are isolated. Thus, the protocol stacks and network processors available from different vendors can be easily integrated with the NPF APIs. The APIs included in the Control Plane Platform Development Kit are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

1.1 Namespace

This document specifies the namespace for the CP-PDK. The namespace is used for locating objects within a compliant system. The system is thought of as a single physical entity such as a chassis containing multiple line cards, a control processor, and an interconnect fabric, but a more comprehensive view of the architecture is provided in the API Framework Reference.

The namespace contains names for compliant and vendor-specific objects and provides the operations necessary for converting between names of such objects and strongly typed methods of accessing and manipulating them. The namespace need not contain objects, or object-specific, or type-specific properties about objects. The implementation of the namespace is left to the system implementer. This document describes the types of functions and semantics required of the namespace.

The goal of the namespace is to provide a convenient set of APIs and naming conventions that can be used to gain access to the system data defined by the NPF and implemented by vendors on network processing platforms. Vendors may also use the namespace to expose new, proprietary services beyond those already defined. Consumers (users) of the namespace API and other CP-PDK APIs are expected to be application writers or box vendors.

Application writers include providers of portable protocol stacks written on top of the APIs, as well as writers of user-level software applications that make inquiries or modifications of the overall system configuration state. Box vendors include providers of networking devices that write control software or integrate third-party application software on top of the APIs in their released systems.

1.2 Assumptions and Dependencies

The namespace, along with the other APIs and services, will be provided in the form of header files and libraries appropriate to the language environment in which they will be used. For compiled language environments (C, C++, and so on), application writers compile against such files and libraries to produce an executable, which runs on the particular network processor platform that was compiled against. If application writers wish to support multiple platforms, they must produce an executable of the appropriate format. Thus, the namespace and other APIs provide compatibility at source code level.

The addition of new services to a network-processing platform (and the associated proprietary additions to the namespace) will be accomplished through source code additions. Such additions could be implemented either by the original platform vendor implementing the APIs or by a third party.

1.3 Terminology

Table 1 lists terms used in this document and provides the expansion of each term.

Table 1. Terminology

Term	Description
Control Element (CE)	In a separated control/data system, refers to the processor(s) responsible for control and configuration of forwarding elements. Used interchangeably with Control Plane (CP).
Control Plane (CP)	See Control Element (CE)
CP PDK	Control Plane Platform Development Kit
Forwarding Element (FE)	In a separated control/data system, refers to the processor(s) responsible for fast path forwarding of data. Used interchangeably with FP.
Forwarding Plane (FP)	See Forwarding Element (FE)
IP	Internet Protocol
MPLS	Multiprotocol Label Switching
NPF	Network Processing Forum
OSAS	Operating System Abstraction Services

Part 2: Namespace Architecture

2 Namespace Architecture

The purpose of the namespace is to allow multiple management applications within a system to identify and group names of managed objects in a hierarchical fashion. The namespace does not logically contain objects. Instead, it contains the names of objects and other system-specific, nonstandard information required to access an object's methods.

2.1 Name Strings and Object Types

Clients of the namespace API interested only in the hierarchical naming layout of the namespace are not ordinarily required to distinguish between the different types of objects it contains. Applications wishing to invoke methods on a particular object must know its type.

A name in the namespace consists of a sequence of 7-bit ASCII characters called a name string. Name strings have no maximum length, although implementations may limit name strings to a maximum of `NPF_NS_PATH_MAX` characters. `NPF_NS_PATH_MAX` should be a system-defined integral constant with a value no less than 256. The special character `/` acts as a hierarchy delimiter. All name strings begin with the hierarchy character. The hierarchy character by itself represents the highest level or root of the hierarchy.

2.2 Referring to Objects

The namespace contains names of objects and the low-level information necessary to invoke methods on them. In addition, the API Framework Reference and API Reference Guides contain definitions of data types used to refer to objects in the system. Combined, several mechanisms are available to refer to objects:

1. Object name – called a name string and implemented by the name space service. May be used to acquire a handle. There is one canonical name per namespace entry, see below.
2. Handle – a process-local reference to an object (generic). May be used to acquire a type-specific handle used in invoking object methods.
3. Alias – an alternative name for an object that has a canonical name (see Section 2.4, Canonical Names and Aliases.)

There are several functions for converting between representations (refer to Section 3, Namespace API.).

2.3 The Namespace Hierarchy

The namespace is a generic tree consisting of vertices and edges rooted at a specially identified root vertex. Each vertex has an identifying name string and may contain zero or more children vertices (immediate descendants of a vertex away from the root along the DAG), as well as zero or one object references. Vertices are known as directories when they contain the names of one or more children vertices. Vertices without children are called leaf vertices or leaves. Objects lacking object references are known as placeholders.

With the exception of the root vertex, all other vertices contain a distinct parent vertex, which is identified by the name of the directory that contains it. The root vertex is its own parent. A directory

may contain any number of children, although an implementation may limit this number to `NPF_NS_MAX`. The integral constant `NPF_NS_MAX` should have a value of 256 or more.

The following diagram shows an instance of the namespace. The solid directional lines show the logical parent-child relationship, and the dotted directional lines show the associate relationship. For example, the node `/System/1/VirtualRouter/1/Interface5` is associated with `/System/1/Blade/0/Port/0`. If just the solid lines are counted, the graph is generic tree. If both solid and dotted lines are counted, the graph is a directional acyclic graph (DAG).

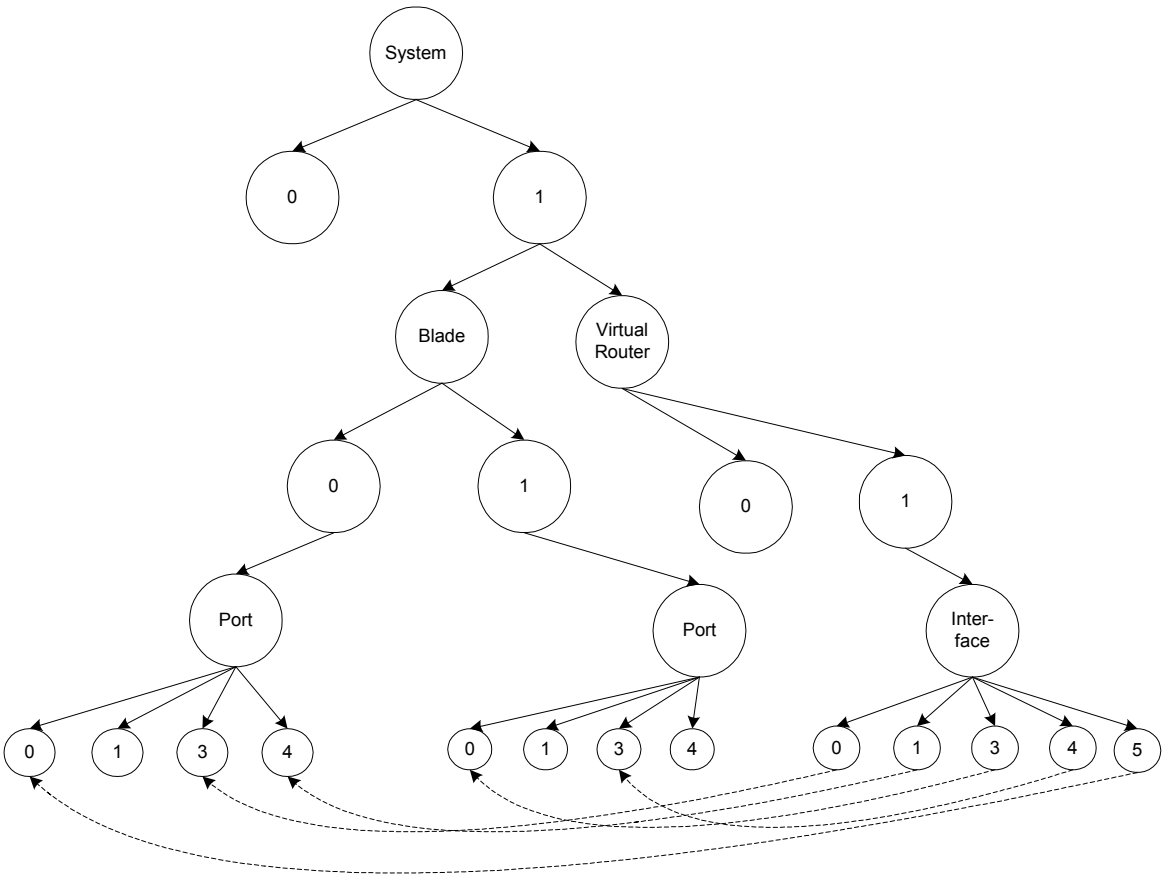

Figure 1. Instance of namespace

2.4 Canonical Names and Aliases

Each vertex identified in the namespace has a specific string that refers to it, which is termed as its canonical name. A namespace vertex may also have zero or more aliases that refer to its canonical name. Aliases and canonical names are not ordinarily distinguished from each other, as both can be used interchangeably within the API to refer to the underlying objects. Special management circumstances like creating or removing an alias are exceptions. Ultimately, the presence or lack of a canonical name within the namespace dictates whether its corresponding object may be referenced through the namespace. If a canonical name is removed from the namespace, its possible illegal aliases may remain, as they are not automatically removed.

2.5 Association

Alias is a type of relationship among the namespace nodes. The primary motivation of the concept of association is to capture the generic many-to-many relationship between ports and interfaces. Here a port is a software representation of a layer-2 network port. An interface is a layer-3 protocol address, such as an IP address.

Since one or many IP addresses can be assigned to one or many ports, the relationship between ports and interfaces is many-to-many. Multiple one-to-many associations represent these many-to-many relationships. In terms of namespace, a set of nodes is associated to another node N. This set of nodes is called associates of N. Another example of association is a port-based VLAN associated with one or more ports.

2.6 Namespace Meta-Data

Each vertex within the namespace is named, and also has a set of associated properties. Vendor-specific extensions required for system operation to these properties are not permitted. This document defines the following properties:

1. **Vertex type**

   Applies to: all vertices

   Contains: an enumerated type, indicating whether vertex is an alias or canonical name

2. **Canonical name**

   Applies to: (leaf) vertices that are aliases

   Contains: the canonical name of the referred-to object

2.7 Namespace Initialization

The namespace does not specify whether it is automatically populated, perhaps by underlying vendor-specific glue code that detects devices, or if it is populated by a management application using the namespace creation APIs defined in Section **Error! Reference source not found.**. In either case, actual discovery of system devices is done in a fashion that is beyond the scope of this document.

2.8 Naming Conventions

Canonical names for objects are created according to a special convention. The convention dictates that names are constructed as a concatenation of pairs (type name, instance number), each separated by the hierarchy character. Thus, a canonical name consists of a repetition of the following:

```
/type name/instance number
```

For example, assume objects of type `IPv4UnicastRouter` contain the names of multiple objects of type `LogicalInterface`, the first of which is called 0. The following name string can be used to refer to this object:

```
/System/0/VirtualView/1/IPv4UnicastRouter/1/LogicalInterface/0
```

The actual string literals `LogicalInterface`, `IPv4UnicastRouter` are not given in this document, but are defined in per-type standards documents. These documents define the name of the type and the containment relationship between types. For example, the fact that `IPv4UnicastRouter` might contain `LogicalInterface`s.

Part 3: Data Structures

3 Data Structures

Please refer to the API Framework Reference for the definitions and semantics of basic data types including NPF_HANDLE and NPF_RET.

The following methods are part of the NPF_MANAGEMENT module and the NPF_NAMESPACE interface.

3.1 Common Data Types

The following are some manifest constants and basic data structures used by namespace both in its API and internal implementation.

NPF_HANDLE	Handle to system data managed by the owning components of the data
NPF_NSHDL	Handle to namespace node managed by the namespace
NPF_NS_PATH_MAX	System constant – maximum length of name string (bytes) for namespace path
NPF_NS_MAXNAME	System constant – maximum length of name string (bytes) for a node
NPF_NS_MAXCHILD	Maximum number of children vertices from a vertex
NPF_RET	Enumerated int return type (see return type section)

3.2 Basic Data Structures

The following is the type of a vertex (node) in the namespace that could be type node, instance node, or alias node.

```
enum NPF_NSNODETYPE{
     TYPE,          // type node
     INSTANCE,      // instance node containing object reference
     ALIAS          // indirect node containing alias
}
```

The following is the entry part of a vertex. The handle to instance node field is ignored if the node is not an alias node. The data handle field is ignored when the node is not an instance node.

```
struct NPF_NS_ENTRY{
   char * name; // null terminated name string for the node
   enum NPF_NSNODETYPE nodeType; // type, instance, alias
   enum NPF_NSDATATYPE dataType;  // blade, port, interface, etc
   union {
      NPF_NSHDL hInstance; // the instance node handle for alias
      NPF_HANDLE hData; // handle to system data for instance
   } entryAttrib;
   int refCount; // reference count to this node
}
```

The following is the directory part of a vertex. This can be potentially large.

```
struct npf_ns_directory {
   int nchildren;
   DList children
}
```

The following is the basic building block of the namespace hierarchy – the vertex that contains: parent, directory part, and entry part. A handle to a namespace node is a pointer to that node.

```
struct NPF_NS_NODE{
   NPF_NSHDL parent; // pointer to the parent node, null for root
   struct npf_ns_directory directory;
   struct npf_ns_entry entry;

   rwlock lock;        // read-write lock for this node
}
```

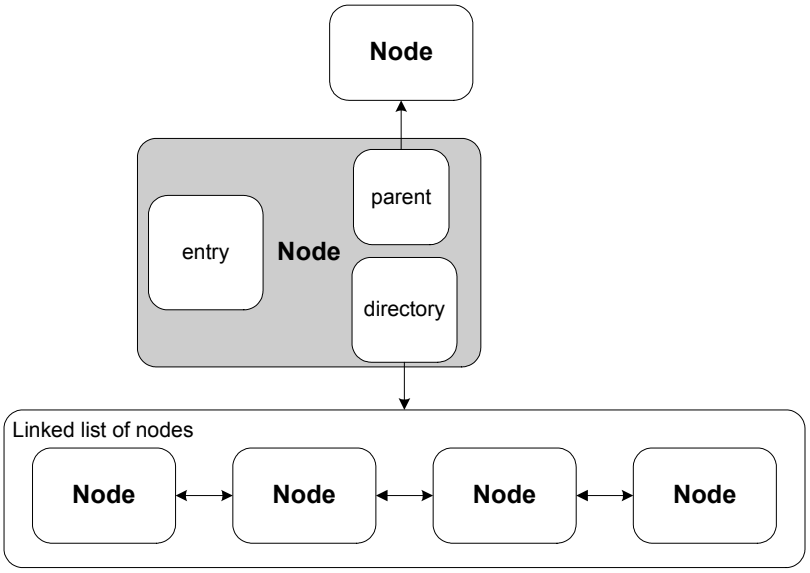

Figure 2. Namespace node linkage

3.3 Return Values

NPF_SUCCESS	Successful return of a function
NPF_INVALID_PARAMETERS	Failed due to invalid input
NPF_OUT_OF_MEMORY	Failed due to out of memory
NPF_NS_NODE_EXISTS	Failed to create a new node, since this node already exists
NPF_NS_NODE_IN_USE	Cannot edit this node because it is in use

NPF_OSAS_FAIL	Failure coming from OS abstraction services layer below the namespace system
NPF_COMPONENT_UNINITIALIZED	Failed due to the namespace not being initialized

Part 4: Namespace API

4 Namespace API

Table 2 lists the namespace functions and provides a brief description of each function. The sections that follow provide detailed descriptions of these functions.

Table 2. Namespace functions

Function	Description
ns_init	Initialize the namespace
npf_ns_create	Create a namespace entry or an alias name
npf_ns_alias	Create an alias for an instance node
npf_ns_delete	Delete an alias or a namespace entry
npf_ns_rename	Change the name string
npf_ns_open	Open a vertex and get a handle to it
npf_ns_getPath	Get the namespace path
npf_ns_getDataType	Get the type of the system data
npf_ns_getDataHandle	Get a handle to the system data
npf_ns_getInstParent	Get the instance parent
npf_ns_getParentNode	Get the parent node
npf_ns_readDir	Retrieve a vertex's directory portion
npf_ns_first	Get a handle to the first child in the directory
npf_ns_next	Get a handle to the next child in the directory
npf_ns_prev	Get a handle to the previous child in the directory
npf_ns_count	Get the number of children in the directory
npf_ns_done	Done with directory iterator
npf_ns_addAssociate	Add an associate
npf_ns_delAssociate	Delete an associate
npf_ns_enumAssociate	Enumerate associates
npf_ns_close	Relinquish a handle
npf_ns_canon	Get the canonical name of a vertex

4.1 Initialize the Namespace

Syntax

```
NPF_RET ns_init()
```

Description

Initialize the namespace.

Input Parameters

None.

Return Values

NPF_SUCCESS	Call successful
NPF_OSAS_FAIL	Failure from the OS Abstraction Services layer

4.2 Add a Namespace Node

Syntax

```
NPF_RET npf_ns_create(          in char  *pathstr,
      in npf_ns_vertex_type     nodeType,
      in npf_data_type          dataType,
      in NPF_NSHDL              hInstance,
      out NPF_NSHDL             *hNode);
```

Description

Create a namespace node at the given path. If the node type is an instance node, the namespace calls the PDK component responsible for the management of this type of data to create the system data and gets the handle to the system data. You can then call `npf_ns_getDataHandle` to translate the namespace handle to the system data handle. The caller can manipulate using set, get, and other data-specific operations the system data based on the data handle. If the node type is alias, the namespace should validate the existence of the instance node aliased by this node.

Input Parameters

pathstr	The null-terminated name string for the path of the node to be created
nodeType	The type of namespace node
dataType	The type of the system data to be referenced by this node if the node is an instance, or by its children node if the node is a type node. This parameter is ignored if the node is an alias node.
hInstance	The handle to the instance node if the node type is alias, ignored otherwise

Output Parameters

`hNode`	Handle to the created node

Return Values

`NPF_SUCCESS`	Call successful
`NPF_OUT_OF_MEMORY`	Call failed due to out of memory
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters
`NPF_NS_NODE_EXISTS`	Node already exists

4.3 Create an Alias

Syntax

```
NPF_RET npf_ns_alias(   in char * canonName,
                        in char * aliasName)
```

Description

Create a namespace alias node for the instance node whose canonical name is provided.

Input Parameters

`canonName`	Null-terminated canonical path name string of the node
`aliasName`	Null-terminated alias path name string of the node

Return Values

`NPF_SUCCESS`	Call successful
`NPF_OUT_OF_MEMORY`	Call failed due to out of memory
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters
`NPF_NS_NODE_EXISTS`	Node already exists

4.4 Delete a Namespace Node

Syntax

```
NPF_RET npf_ns_delete(in char * name)
```

Description

If the path string is an alias, remove the specified alias node. If the path string is the canonical name, remove the instance node and preserve all aliases associated with this vertex. A node can only be deleted there is no open handle to the node and its descendant nodes. If a node is deleted, all of its children are also deleted.

Input Parameters

`name`	Null-terminated canonical path name string

Return Values

`NPF_SUCCESS`	Call successful
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters
`NPF_NS_NODE_IN_USE`	The node is currently in use

4.5 Rename a Namespace Node

Syntax

```
NPF_RET npf_ns_rename(  in char * oldName,
                        in char * newName)
```

Description

Change the name of a node. This may result in moving a node from one location on the hierarchy to another location. These names are either canonical names, or alias names. One cannot change a canonical name to an alias name or vice versa. Note that an alias node to an instance node need not be updated if the canonical name of the instance node is changed, since the alias node refers to the instance node by namespace handle, not by path name string.

Input Parameters

`oldName`	Null-terminated old path name string of the node
`newName`	Null-terminated new path name string of the node

Return Values

NPF_SUCCESS	Call successful
NPF_OUT_OF_MEMORY	Call failed due to out of memory
NPF_COMPONENT_UNINITIATED	Namespace is not initiated
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_NS_NODE_EXISTS	Node already exists

4.6 Get a Handle to a Node

Syntax

```
NPF_RET npf_ns_open(    in char *      name,
                        out NPF_NSHDL * hNode)
```

Description

Open a node and output a handle to the namespace node.

Input Parameters

name	Null-terminated path name string of the node

Output Parameters

hNode	Handle to the node

Return Values

NPF_SUCCESS	Call successful
NPF_COMPONENT_UNINITIALIZED	Namespace not initialized
NPF_INVALID_PARAMETERS	Invalid input parameters

4.7 Get Path by Handle

Syntax

```
NPF_RET npf_ns_getPath( in NPF_NSHDL hNode,
      out   char * path);
```

Description

Get the namespace path from the namespace handle of a node. The caller is responsible for allocating memory of size NPF_NS_PATH_MAX for the path string before calling this function.

Input Parameters

hNode	Namespace handle to a node

Output Parameters

path	Null-terminated path string of the node. The caller should allocate memory of size NPF_NS_PATH_MAX for this string before calling this function.

Return Values

NPF_SUCCESS	Call successful
NPF_COMPONENT_UNINITIALIZED	Namespace not initialized
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_OUT_OF_MEMORY	Call failed due to out of memory

4.8 Get Data Type

Syntax

```
NPF_RET npf_ns_getDataType(   in NPF_NSHDL   nh,
        out DataType * dataType);
```

Description

Reads the type of system data referenced by the argument namespace handle nh.

Input Parameters

nh	Handle to the node

Output Parameters

dataType	Output data type

Return Values

NPF_SUCCESS	Call successful
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_OUT_OF_MEMORY	Call failed due to out of memory

4.9 Get Data Handle

Syntax

```
NPF_RET npf_ns_getDataHandle( in NPF_NSHDL    nh,
                              out NPF_HANDLE * hData);
```

Description

Get the handle to the system data referenced by a node. The function returns an invalid input error if the node is not an instance node.

Input Parameters

nh	Handle to the node

Output Parameters

hData	Output data handle

Return Values

NPF_SUCCESS	Call successful
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_OUT_OF_MEMORY	Call failed due to out of memory

4.10 Get Instance Parent

Syntax

```
NPF_RET npf_ns_getInstParent( in NPF_NSHDL     nh,
                              out NPF_NSHDL * hInstParent);
```

Description

Get an open handle to the instance parent of an instance node nh, which is the grandparent in the namespace hierarchy. The caller must close the open handle after using it. The function returns an invalid input error if the node is not an instance node.

Input Parameters

nh	Open handle to an instance node

Output Parameters

hInstParent	Output open handle to the instance parent

Return Values

NPF_SUCCESS	Call successful
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_COMPONENT_UNINITIALIZED	Namespace not initialized

4.11 Get Parent in Namespace

Syntax

```
NPF_RET npf_ns_getParentNode( in NPF_NSHDL   nh,
                              out NPF_NSHDL * hParent);
```

Description

Get an open handle to the parent node in the namespace hierarchy. The caller must close the handle after using it.

Input Parameters

nh	Open handle to the node

Output Parameters

hParent	Output open handle to the parent node

Return Values

NPF_SUCCESS	Call successful
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_COMPONENT_UNINITIALIZED	Namespace not initialized

4.12 Get Node Directory Info

Syntax

```
NPF_RET npf_ns_readDir( in char *       name,
                        out NODE_ITERATOR * dirItr)
```

Description

Retrieve the directory information of a node. This allows you to access the children nodes of the current node. NODE_ITERATOR is an iterator of a set of ordered namespace nodes, in this case, the ordered set of children. The functions used to traverse or count the set of nodes are:

```
NPF_NSHDL npf_ns_first(NODE_ITERATOR dirItr);
NPF_NSHDL npf_ns_next(NODE_ITERATOR dirItr);
NPF_NSHDL npf_ns_prev(NODE_ITERATOR dirItr);
int npf_ns_count(NODE_ITERATOR dirItr);
```

```
void npf_ns_done(NODE_ITERATOR dirItr);
```

`npf_ns_first()` returns an open handle to the first node in the directory, `npf_ns_next()` returns an open handle to the next node, and `npf_ns_prev()` returns an open handle to the previous node. A return of `0` signals the end of the iteration. The client must call `npf_ns_close()` for each of the open handles and `npf_ns_done()` after using the iterator.

Input Parameters

`name`	Null-terminated path name string of the node

Output Parameter

`dirItr`	An iterator of the directory

Return Values

`NPF_SUCCESS`	Call successful
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters
`NPF_OUT_OF_MEMORY`	Call failed due to out of memory

4.13 Get Handle to First Child Node

Syntax

```
NPF_NSHDL npf_ns_first (in NODE_ITERATOR dirItr)
```

Description

Returns an open namespace handle to the first child in the directory. The client must close this handle after using it.

Input Parameters

`DirItr`	A directory handle

Return Values

Open namespace handle to the first child, or `NULL` if none are available.

4.14 Get Next Node Iterator

Syntax

```
NPF_NSHDL npf_ns_next(in NODE_ITERATOR nodeItr)
```

Description

Get an open namespace handle to the next child in the directory. The caller must close the handle after using it.

Input Parameters

`nodeItr`	An open handle on the node

Return Values

Handle to next child, or `NULL` if child does not exist.

4.15 Get Previous Node Iterator

Syntax

```
NPF_NSHDL npf_ns_prev(in NODE_ITERATOR nodeItr)
```

Description

Get an open namespace handle to the previous child in the directory. The caller must close the handle after using it.

Input Parameters

`nodeItr`	An open handle on the node

Return Values

Handle to previous child, or `NULL` if child does not exist.

4.16 Get Number of Children

Syntax

```
int npf_ns_count (in NODE_ITERATOR nodeItr)
```

Description

Returns the number of children in the directory.

Input Parameters

`nodeItr`	An open handle on the node

Return Values

Number of children in the directory, or `-1` if the call is not successful.

4.17 Close Directory

Syntax

```
void npf_ns_done(in NODE_ITERATOR nodeItr)
```

Description

Removes directory node after the client has finished using it.

Input Parameters

`nodeItr`	An open handle on the node

4.18 Add an Associate

Syntax

```
NPF_RET npf_ns_addAssociate(  in NPF_NSHDL hNode,
                              in NPF_NSHDL hAssociate)
```

Description

This function adds an associate to the namespace node.

Input Parameters

`hNode`	Handle to a namespace node
`hAssociate`	Handle to the node to be associated with `hNode`

Return Values

`NPF_SUCCESS`	Call successful
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters

4.19 Delete an Associate

Syntax

```
NPF_RET npf_ns_delAssociate(  in NPF_NSHDL hNode,
                              in NPF_NSHDL hAssociate)
```

Description

This function deletes an associate from the namespace node.

Input Parameters

`hNode`	Handle to a namespace node
`hAssociate`	Handle to the associated node

Return Values

`NPF_SUCCESS`	Call successful
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters

4.20 Enumerate Associates

Syntax

```
NPF_RET npf_ns_enumAssociate( in NPF_NSHDL      hNode,
                              out NODE_ITERATOR associateItr)
```

Description

This function enumerates associates of a namespace node. The interface of `NODE_ITERATOR` is described in Section 4.12.

Input Parameters

`hNode`	Handle to a namespace node

Output Parameters

`associateItr`	Node iterator for the associates

Return Values

`NPF_SUCCESS`	Call successful
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters
`NPF_OUT_OF_MEMORY`	Call failed due to out of memory

4.21 Close a Node Handle

Syntax

```
NPF_RET npf_ns_close(in NPF_NSHDL nh)
```

Description

Close an open handle on a node. After using an open handle to a node obtained from `npf_ns_create()`, `npf_ns_open()`, `npf_ns_first()`, `npf_ns_next()`, `npf_ns_getInstParent()`, or `npf_ns_getParentNode()`, the handle must be closed with this function.

Input Parameters

`nh`	Open handle on the node

Return Values

`NPF_SUCCESS`	Call successful
`NPF_COMPONENT_UNINITIALIZED`	Namespace not initialized
`NPF_INVALID_PARAMETERS`	Invalid input parameters

4.22 Get Canonical Node Name

Syntax

```
NPF_RET npf_ns_canon(   in  char * namestring,
                        out char * canon_name)
```

Description

Given a name string (usually an alias), provide the canonical name. The function returns an error if the provided name string is a dangling alias name. The caller is responsible for allocating memory of size `NPF_NS_PATH_MAX` before calling this function.

Input Parameters

`namestring`	Null-terminated path name string of the node

Output Parameters

`canon_name`	Canonical path name string of the node

Return Values

NPF_SUCCESS	Call successful
NPF_COMPONENT_UNINITIALIZED	Namespace not initialized
NPF_INVALID_PARAMETERS	Invalid input parameters
NPF_OUT_OF_MEMORY	Call failed due to out of memory