Namespace

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

intel

Contents

Tables

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Anantha Rathnam
2.1	Updated for Release 2.1	December 2003	Anantha Rathnam
2.0	Updated for Release 2.0	August 2003	Anantha Rathnam

Part 1: Introduction

1 Introduction

Network elements such as switches and routers can be classified into three logical operational components:

- Control plane
- Forwarding plane
- Management plane

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

The forwarding plane makes decisions based on this information and 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, and performs logging.

The introduction of standardized Application Program Interface (API) within the above-mentioned planes can help system vendors, Original Equipment Manufacturer (OEM), 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.

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 (CP-PDK) are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

This document specifies the internal design of the namespace component of the CP-PDK. This includes the description and design of the main internal data structures as well as algorithms used within the component.

1.1 Terminology

Table 1 lists terms used in this document and provides an expansion for each term.

Table 1. Terminology table

Term	Description
CLI	Command-Line Interface
C&M	Configuration and Management Module of CP-PDK

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 CP.
Control Plane (CP)	See Control Element (CE).
ForCES	Forwarding and Control Element Separation protocol
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).
IDB	Information Database
NPF	Network Processing Forum
PDK	Platform Development Kit
SNMP	Simple Network Management Protocol

1.2 References

Table 2 lists documents referenced in, or related to, this document.

Table 2. Reference table

Reference	Document Name
[1]	NPF Application Level API Framework
[2]	CP-PDKIPv4 API Reference
[3]	NPF Namespace Specification
[4]	CP-PDK Software Architecture Overview
[5]	NPF Classification API document
[6]	NPF Configuration Application and Information Database document
[7]	CP-PDK API Framework Reference
[8]	CP-PDK Configuration and Management API Reference
[9]	Platform Namespace API Reference

Part 2: Overview

2 Overview

The namespace is an in-memory directory service used by components in an NPF-compliant system to locate system data. An NPF-compliant system is a network element that supports the separation of the control plane from the data-forwarding plane as defined in [1]. A well-defined application API called the NPF API is placed in the framework, which provides the access to the basic system resources to upper level services. The CP-PDK exposes the NPF API. The namespace is a module residing in the CP-PDK.

2.1 Requirements

Many components inside an NPF-compliant system need basic system data such as software representation of forwarding blades and network interfaces installed on the forwarding blades. The namespace provides the directory service to these components to locate the basic system data. This directory service is provided through the namespace API that is neutral to the basic system data. A component first locates the system data through the namespace, and then it deals with the system data specific semantics with the individual data.

2.2 High-Level Functionality Overview

The referencing data stored in the namespace is structured based on a certain data model called the namespace schema. The definition of each schema is the task of other groups. This design document focuses on the basic services provided by the namespace that are used by various versions of the namespace schema.

Figure 1 shows the architecture of an NPF-compliant system that includes the CP-PDK and shows where the namespace fits in.

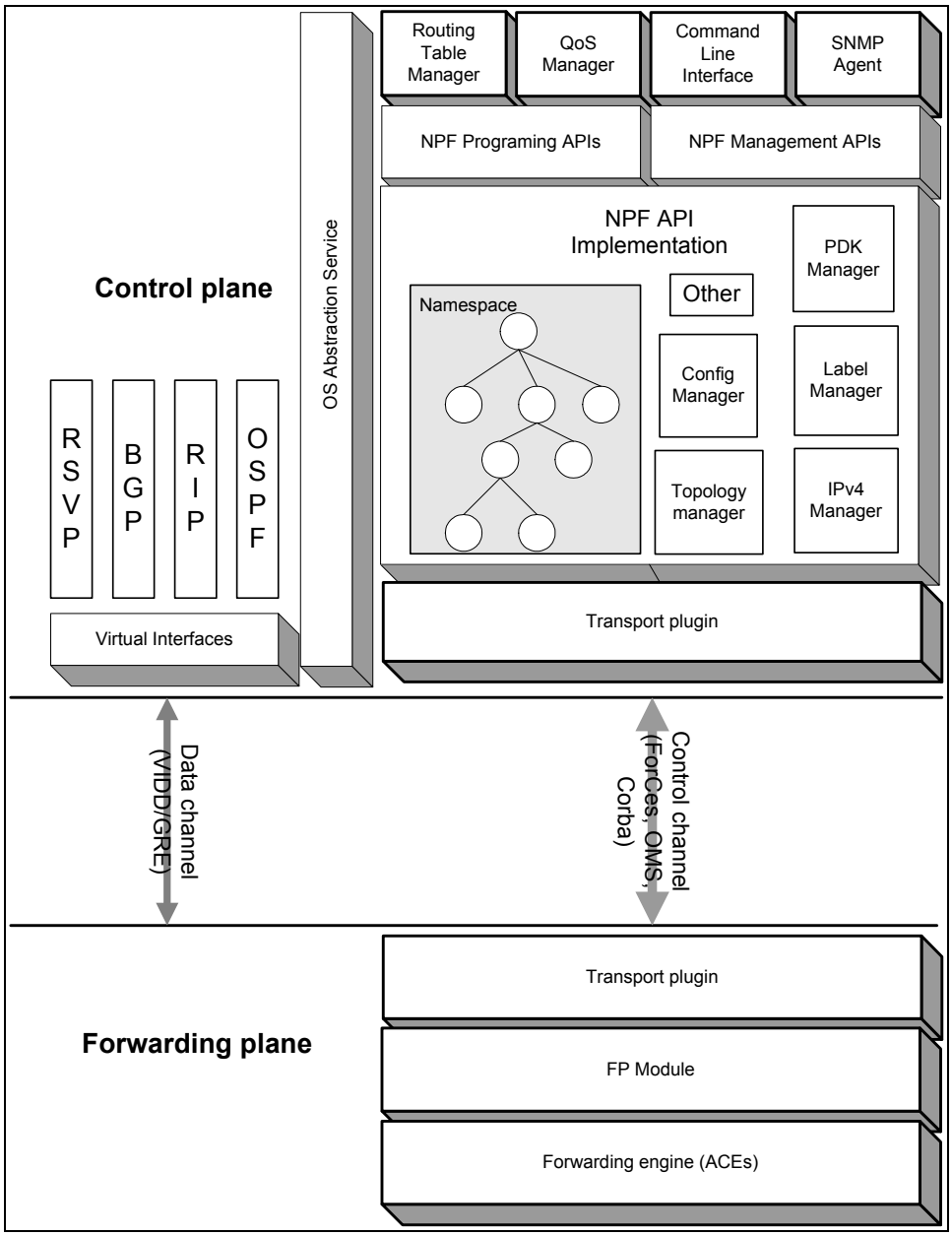

Figure 1: CP-PDK architecture

2.3 Design Considerations

A directory service is provided through the namespace API that is neutral to the basic system data. It should maintain the lifespan of the system data:

- It creates reference counts

- maintains reference counts
- deletes system data

As various PDK components call into the namespace, the operation should be synchronized and optimized.

Part 3: Namespace Design

3 Namespace Design

3.1 High-Level Overview

The namespace is a hierarchical in-memory directory. According to the NPF namespace specification, a node or entry of this hierarchy is assigned with a name string of the regular expression pattern:

```
(/typeName/instanceNumber)*(/typeName[/instanceNumber])
```

An example of such a name string is `/NpfSys/0/FE/0/port/2`, which refers to the second port installed on the first forwarding blade of the first system (`/NPFSys/0`). Each node of the namespace has a canonical name string and zero or more alias name strings.

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 alias relationship. For example, the node `/NPFSys/0/Interface/5` is an alias to `/NPFSys/0/FE/0/Port/4`. If only the solid lines are counted, the graph is a generic tree. If both solid and dotted lines are counted, the graph is a directional acyclic graph.

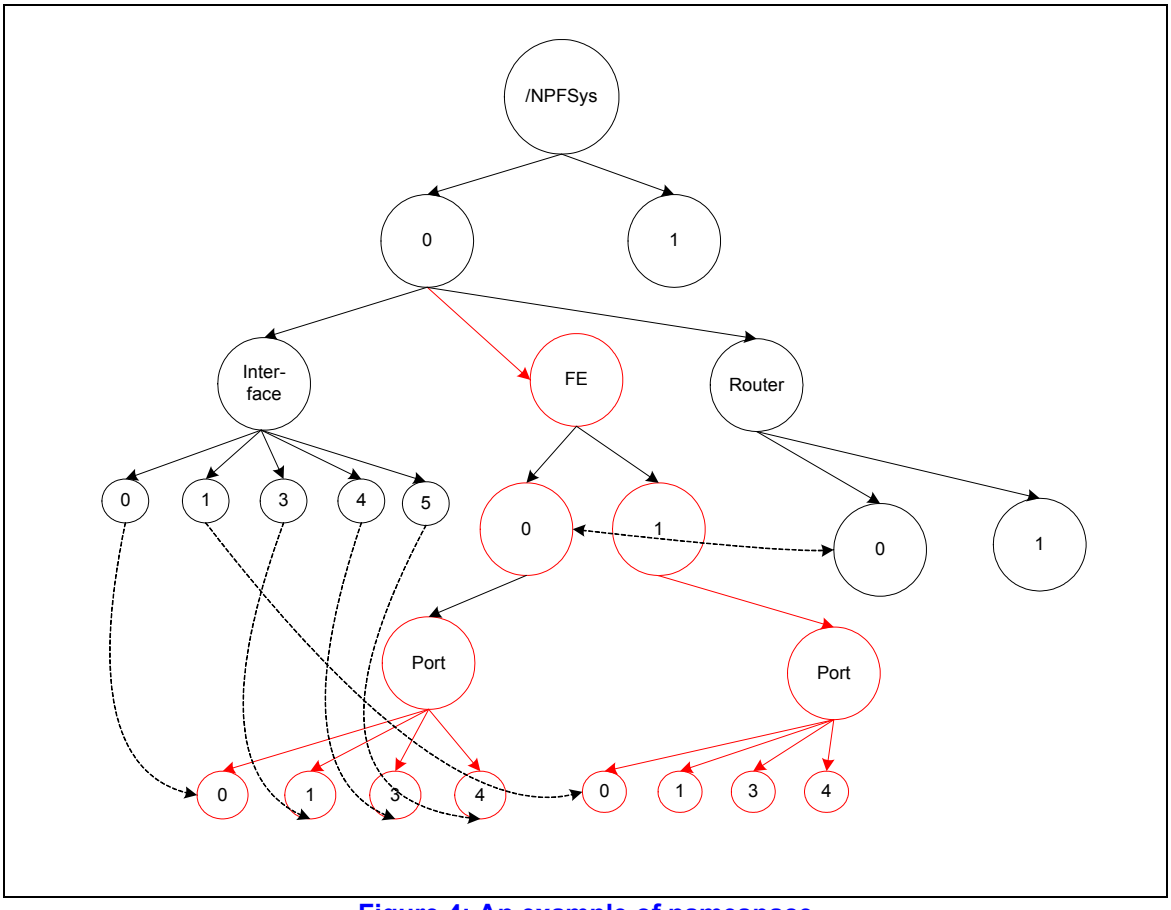

Figure 4: An example of namespace

There are three kinds of nodes in a namespace:

1. Bears the type name, such as NPFSys and FE. This kind of nodes is called a type node.
2. Node is for the instances of certain types. They are called instance nodes. The detailed instance specific information is not stored within the instance node. Such information is stored outside the namespace and referenced by the instance node. For example, the instance node of `/NPFSys/0/FE/0/Port/0` of the namespace provides a reference to the data structure as a software representation of the physical port. Here the data structure for the port is not located inside the namespace, and is maintained by other components, such as the configuration manager of the PDK.
3. Is called an alias node, which represents an alias of an instance node. An alias node does not contain the reference to the system data; one has to go to the instance node to get the reference to the system data.

3.2 External API

The following table provides a summary of the namespace interface. For a complete description of each function, refer to the Control Plane PDK Platform Namespace API Reference.

Table 3. External API table

Function	Description
ns_init	Initialize the namespace
npf_ns_create	Create an namespace entry or an alias name
npf_ns_alias	Create an alias for an instance node
npf_ns_delete	Delete an alias or an namespace entry
npf_ns_rename	Change the name string to a new one
npf_ns_open	Open a node to get an handle to it
npf_ns_getPath	Get the path string to an instance node
npf_ns_getDataType	Get the type of the system data
npf_ns_getDataHandle	Get a handle to the system data
npf_ns_getParentNode	Get a handle to the parent node
npf_ns_getInstParent	Get a handle to the instance parent node
npf_ns_readDir	Retrieve a node's directory portion
npf_ns_addAssociate	Add an associate to a node
npf_ns_delAssociate	Delete an associate from a node
npf_ns_enumAssociate	Enumerate the associates of the node

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_close	Relinquish handle
npf_ns_canon	Get the canonical name of a node

3.3 Implementation Details

This section provides information on implementation details. This section also explains major data structures, threading and synchronization, and algorithm explanation.

3.3.1 Major Data Structures

The following are some manifest constants and basic data structures used by the 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 – max length of name string (bytes) for namespace path

`NPF_NS_MAXNAME` System constant – max length of name string (bytes) for a node

`NPF_NS_MAXCHILD` Max number of children nodes from a node

`NPF_RET` Enumerated (int) return type

1. First, the type of a node in the namespace could be type node, instance node, or alias node:

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

2. Next is the entry part of a node. The handle to the instance node field is ignored if the node if 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
}
```

3. Next is the directory part of a node. This can potentially be large.

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

4. Next is the basic building block of the namespace hierarchy; a node that contains parent, directory part, and entry part. A handle to a namespace node is just 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
}
```

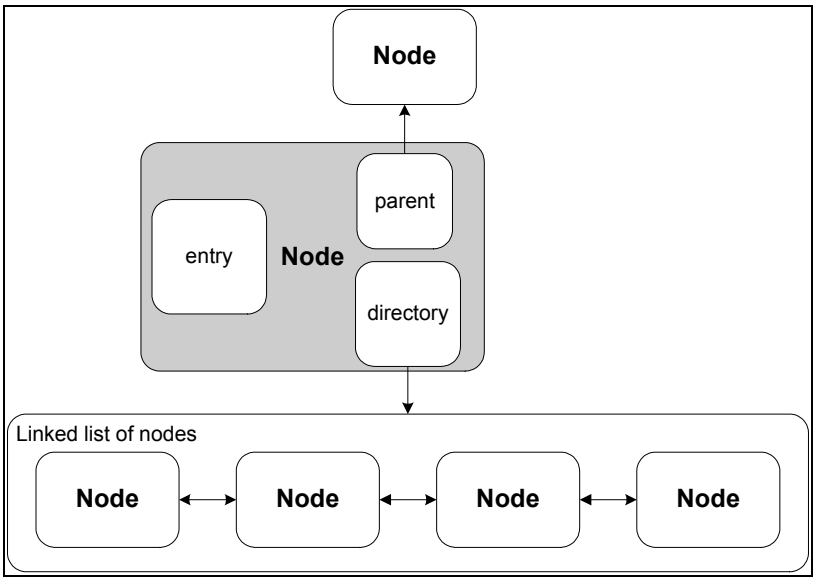

Figure 5: Namespace node

3.3.2 Threading and Synchronization

The namespace is a merge point of many operations in an NPF system, as various components call into it to get the handles they need for their operations. It is important to make it thread safe in an efficient way.

In a simple design, we have one lock (a mutex) for both read and write operations on the entire namespace. When one thread reads or writes to a node of the namespace, it locks all other threads out of the namespace. This lock is initialized during an `npf_ns_init` call and cleaned up during shutdown.

The granularity of this global mutex is not optimized for performance. A version of an optimized thread safety mechanism is described in the Section [4]. This design has the following features.

- Single-lock semantics are exposed to the clients of the locking mechanism, which are mostly implementations of namespace interfaces. That minimizes the effort of the client thread to protect the namespace.
- When a thread reads or writes to a node of the namespace, the portion of the namespace locked by this thread is minimized, so the chance for other threads to use the rest of the namespace is maximized.
- This lock treats read and write operations differently, so the chance for concurrent operations is maximized while protection is enforced.

This mechanism is called read/write lock for hierarchy. In its implementation, multiple mutexes are used in a carefully coordinated manner so that the performance optimization and the single lock semantics are provided.

3.3.3 Algorithm Explanation

Populate the Namespace

The namespace is populated during the start-up of the system by the PDK configuration application.

1. The configuration application reads the initial system configuration information from a persistent media called Information Database (IDB).
2. Then it creates the namespace node by providing the type info of the system data.
3. Based on the type info, the namespace node finds the PDK component responsible for management of this system data, typically the Configuration Manager (CM).
4. Then the namespace calls the CM to create the system data and get a handle to the data.
5. The namespace returns the handle to the namespace node in the open mode.

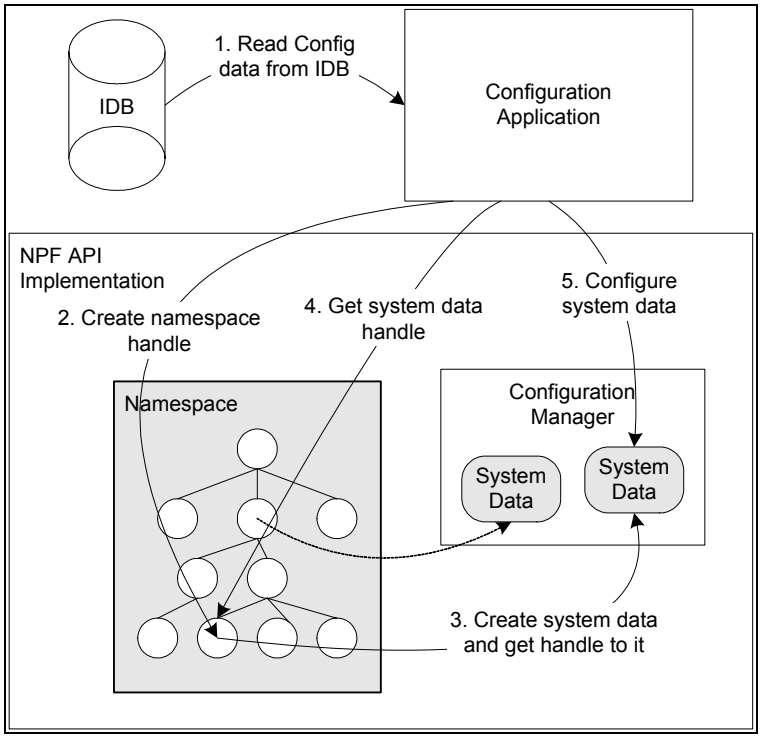

Figure 6: Populate the namespace

Next, the configuration application gets the handle to the system data from the handle to the namespace node. Finally, it configures the system data through the handle based on the configuration information read from the IDB. Finally, it closes the namespace handle. The configuration application repeats these steps node-by-node from top down in depth first order.

Configure System State

In a typical scenario where a system state is changed administratively, the administrative configuration application, e.g., a CLI, a policy agent, or an SNMP agent:,

1. It first calls the namespace API to open a handle to the namespace node and get the handle to the system data that is managed by the configuration management module.
2. Then it calls the CM API to change the state of the system data. The CM module talks to the FE through the FP Plugin API.
3. When the change is made, successfully or unsuccessfully, in the FE, the FP Plugin API calls the callback function registered by the CM earlier.

During the callback, the CM updates the state of the system data to reflect the change in the FE and informs the CLI about the completion. Finally, the CLI closes the handle to namespace node.

This description only uses the CLI as an example of the modules that change the state of system data. The above description does not use any specific properties of the CLI, so any other module, either inside or outside the PDK, can follow the same method to change the state of system data during runtime.

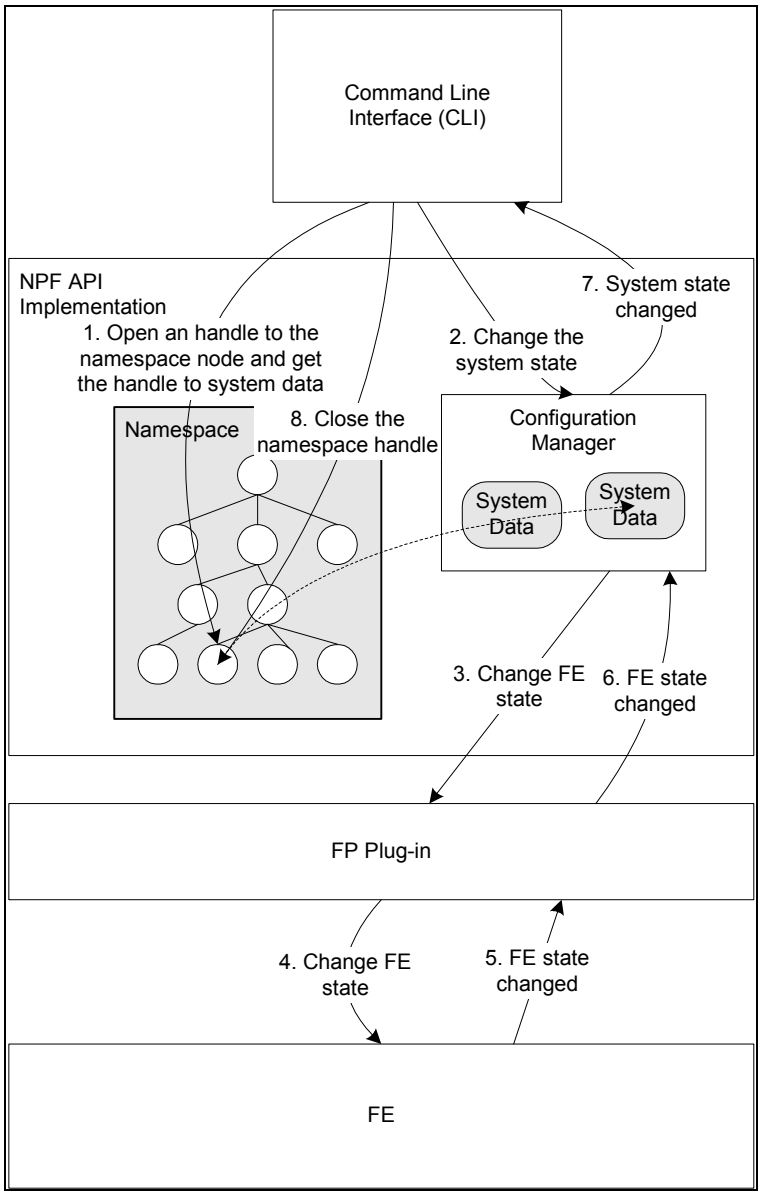

Figure 7: Change system state administratively

Clean Up Namespace

The system is shut down by a PDK application, such as the configuration application.

1. The configuration application first opens a namespace node and gets the handle to the system data from the node.
2. It closes and deletes the namespace node.

The other components of the PDK should be shut down first so that all handles to the namespace nodes are closed. The configuration application repeats these steps from bottom up. The process is the reverse of the population of the namespace.

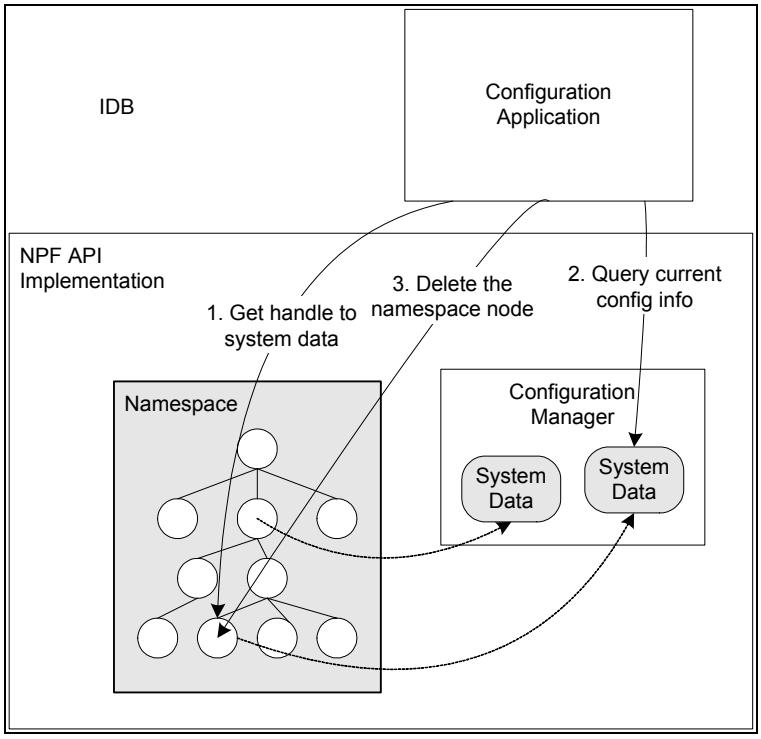

Figure 8: Clean up the namespace

3.4 Modularity

Based on the above interaction analysis, the namespace requires the PDK components that are responsible for management of various system data in the following aspects:

1. When namespace initializes the system data, the components should allocate the memory space for the system data and return a handle to the namespace. At this moment, the system data is initialized to its default values. Later, the configuration application sets them to the configured values.
2. All access to the system data must start with an open handle to the corresponding namespace node and end with closing that handle. In between, the client, either a PDK application or an internal component, should map this namespace handle to the handle to the system data (see `npf_ns_getdatahandle`). In this way, the namespace can maintain the outstanding reference count on the system data.
3. The component should protect its own system data by a per-component lock. The lock used by the namespace only protects the namespace hierarchy and its nodes. See Section 4 for more details on this. The set of per-component locks is organized globally to avoid deadlock. The way they are organized is by introducing a global order to them. See the *API Framework Reference* [7] for details.
4. Callback registrations are made with the management components of the system data. These components will call the callback when the corresponding FP configuration calls are completed or interesting events occur at the FP.

A client of the namespace gets a handle to the system data from the namespace. If the client must access the data directly through pointers instead of a handle, it is the managing component's responsibility to expose the pointer and protect the data pointed to by the pointer.

Part 4: Read/Write Lock Mechanism

4 Read/Write Lock Mechanism

This chapter describes the read-write lock for thread safety of a hierarchy.

The read/write lock mechanism designed in this section exposes the following interface, which has the single-lock semantics.

Table 4. Lock mechanism

Functions	Description
Bool Lock (int operation, Node N)	Acquire the lock for an operation on node N. The operation could be READ or WRITE.
Bool Unlock (Node N)	Release the lock from the node.

The read/write lock has the following behaviors.

1. When reading a namespace node N, the root path of N (all the nodes along the path from the root node to N, including N) is locked for read.
2. When writing to a node N, the root path of N minus N is locked for read and N itself is locked for write.
3. You can read a node N if and only if there is no active write on N.
4. You can write to a node N if and only if there is no active read or write on N.

These behaviors make it possible to lock only the minimal set of nodes for read/write operations at a node and discriminate write operations from read operations to maximize the concurrency while providing thread safety.

Our design is based on the GNSS Navigation Unit (GNU) `rwlock` implemented for p-thread, which has the behaviors similar to 3 and 4. We describe `rwlock` in the following pseudo-code. This code fixes a bug in the GNU rwlock implementation, where a continuous stream of readers can lock all writers out forever.

```
struct rwlock{
            int state;  // if the lock is initialized
            int mode;   // ongoing operation is READ or WRITE
int nr;     // number of active readers
int nw;     // number of outstanding writers
            mutex r;    // read mutex
            mutex w;    // write mutex
}
```

The error checking for initialization of the mutex is skipped in the following function.

```
bool Init(rwlock lock)
{
            lock.state = INITIALIZED;
            lock.nr = 0;
            Init(r);
            Init(w);
```

```
            return true;
}
```

The error checking for the failure of `wait` and `signal` in the following two functions is skipped, as is state checking.

```
bool Acquire(rwlock lock, int op)
{
            if(op == WRITE)
{
                  nw++;
wait(w);
                  mode = WRITE;
            }
            else
            {
                  wait(r);
                  nr++;
                  if(nr == 1 | nw > 0)    wait(w);
                  mode = READ;
                  signal(r);
            }
            return true;
}
```

The blue code segment says that:

"I start to read (`nr++`) and if I am the first reader (`nr == 1`) or there is an outstanding writer (nw > 0), wait for the writer to finish or lock the incoming writers out (`wait(w)`)".

```
bool Release(rwlock lock)
{
            if(mode == WRITE)
{
signal(w);
nw--;
            }
            else
            {
                  wait(r);
                  nr--;
                  if(nr == 0 | nw > 0) signal(w);
                  signal(r);
            }
            return true;
}
```

The red code segment says that:

" I finish reading (`nr--`) and if I am the last reader (`nr == 0`) or there is an outstanding writer (nw > 0), let the writer in (`signal(w)`)".

The design is extended from a single node of data to a generic hierarchy that has m nodes. When a node is added into the hierarchy, we initialize an rwlock to associate with the node. Every node N in the tree has a unique root path, which is the ordered set of nodes starting from the root node to the node N, including N. We denote the root path of N by Path (N). The inverse path, which is the root path in the reverse order is denoted by $Path^{-1}$ (N). The notion Path (N)\N stands for the root path minus the node N.

```
bool Lock(int op, Node N)
{
            if(op == WRITE)
            {
                  for each node X in Path(N)\N
                  {
                        if(Acquire(rwlock of X, READ) == false)
                        {
                              for each node Y in Path-1(X)
                                    release(rwlock of Y);
                              return false;
                        }
                  }
                  if(Acquire(rwlock of N, WRITE) == false)
                  {
                        for each node Y in Path-1(N)\N
                              release(rwlock of Y);
                        return false;
                  }
            }
            else
            {
                  for each node X in Path(N)
                  {
                        if(Acquire(rwlock of X, READ) == false)
                        {
                              for each node Y in Path-1(X)
                                    release(rwlock of Y);
                              return false;
                        }
                  }
            }
            return true;
}
bool Unlock(Node N)
{
```

```
            for each node X in Path-1(N)
                  Release(rwlock of X);
            return true;
}
```

When the hierarchy has only a single node, the pseudo code shown degenerates to the GNU rwlock. It should be noted that the acquire and release functions are not made atomic. They can fail in the middle of walking through the path. Like the rwlock implementation, the wait and signal functions on a mutex can fail. The right way to use this lock mechanism is as follows:

```
If(Lock(operation, N)) {
            do the read/write operation;
Unlock(N);
}
```

It is the do the read/write operation processing made atomic by the locking protection that matches the design goal.