# API Framework

## 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 | 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: 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/.

This document describes the Control Plane PDK framework design and common issues. The PDK is designed in C language with object-oriented abstraction of forwarding engine functionality. Each component provides sets of APIs to operate and manage the forwarding plane.

The Configuration and Management and GTP APIs use the data structures and function signatures presented in this document. All other modules, including IPv4, IPv6, IPv6TM, MPLS, DiffServ, IntServ and ATM use conventions specified in the NPF Conventions document.

The PDK is also designed for a multi-thread-safe environment. It is linked as a shared library to all the application threads/processes, so applications can invoke multiple PDK API calls simultaneously. Keep in mind that applications must still enforce the proper critical sections if their PDK API calls contain dependencies.

Figure 1 illustrates the CP-PDK architecture.

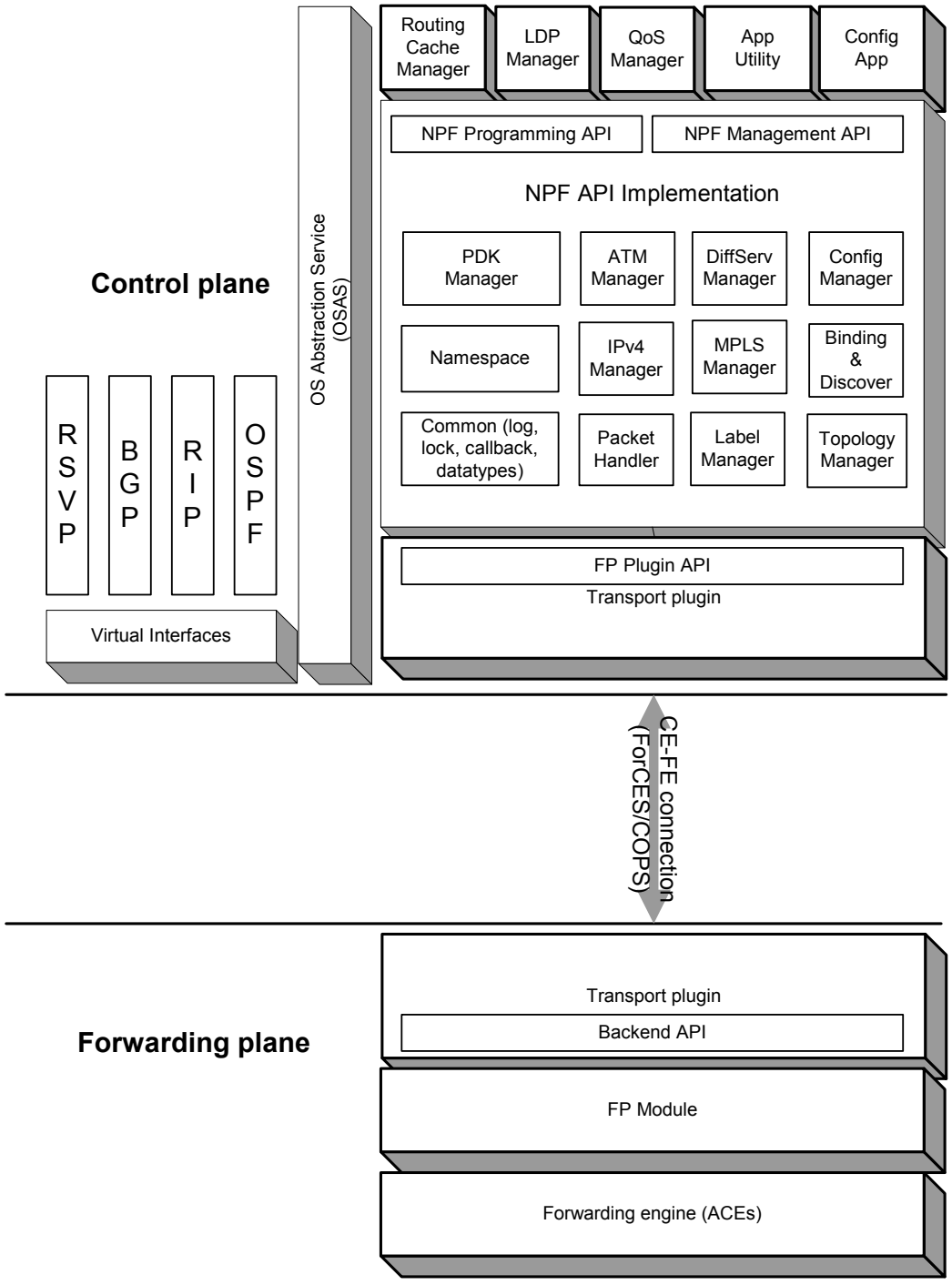



**Figure 1: CP-PDK architecture**

## 1.1 Terminology

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

**Table 1. Terminology**

| Version | Description |
|---|---|
| API | Application Programming Interface |
| CE | Control Element |
| COPS | Common Open Policy Service |
| CP | Control Plane |
| DiffServ | Differentiated Services |
| FE | Forwarding Element |
| ForCES | Forwarding and Control Element Separation protocol |
| IntServ | Integrated Services |
| IXA | Internet Exchange Architecture |
| NPF | Network Processing Forum |
| PDK | Platform Development Kit |

## 1.2 Assumptions and Dependencies

The following assumptions apply to CP-PDK:

- The APIs assume the existence of a compliant namespace present on the system
- It is the responsibility of the PDK to handle the packet flows traveling over multiple blades inside a single ON router. The application running on top of the APIs is hidden from the fact of multiple forwarding planes.
- The APIs operates on handles. Handles to objects are assigned in the manner required by the Namespace API.
- Most functions of the API are asynchronous. Applications have to register callbacks for any subset of the API calls. The callback mechanism and usage model is described in the subsequent section.
-

# *Part 2: Common Types*

# 2 Common Types

## 2.1 Object Types

### 2.1.1 IPA

**Definition**

```
typedef uint32_t IPA
```

**Description**

Common data type for IP address.

### 2.1.2 npf_ipaddr_addr2char

**Signature**

```
int npf_ipaddr_addr2char(  struct in_addr in,
                                    char *buff  )
```

**Description**

Converts the binary `in_addr` to number-and-dot format.

**Input Parameters**

```
struct in_addr or npf_ipaddr
```

**Output Parameters**

```
buff = 111.112.113.114
```

**Return Values**

```
NPF_SUCCESS
NPF_INVALID_PARAMETERS
```

### 2.1.3 HWADDR

**Definition**

```
typedef unsigned char HWADDR[MAX_ADDR_LEN]
```

**Description**

Common data structure for MAC address.

### 2.1.4 npf_hwaddr_addr2char

**Signature**

```
int npf_hwaddr_addr2char(  char *ptr,
                           char *buff  )
```

**Description**

Converts an ethernet address to readable format.

**Input Parameters**

```
unsigned char[6]
```

**Output Parameters**

```
buff = 00:AA:BB:CC:DD:EE
```

**Return Values**

```
NPF_SUCCESS
NPF_INVALID_PARAMETERS
```

### 2.1.5 npf_hwaddr_char2saddr

**Signature**

```
int npf_hwaddr_char2saddr(  char *bufp,
                                    struct sockaddr *sap  )
```

**Description**

Inputs an ethernet address and converts to binary.

**Input Parameters**

```
bufp = "00:AA:BB:CC:DD:EE" or "00AABBCCDDEE"
```

**Output Parameters**

```
sap->sa_family = ARPHRD_ETHER;
sap->sa_data = unsigned char[6] type
```

**Return Values**

```
NPF_SUCCESS
NPF_INVALID_PARAMETERS
```

## 2.2 Data Types

### 2.2.1 PTR_UNDEFINED

**Definition**

```
#define PTR_UNDEFINED   0xFFFFFFFF
```

**Description**

Used for all undefined pointers.

## 2.2.2 NPF_DATATYPE

**Definition**

```
enum {   NPF_SYSTEM,
         NPF_IPROUTER,
         NPF_ROUTETABLE,
         NPF_ARPTABLE,
         NPF_FTNTABLE,
         NPF_ILMTABLE,
         NPF_FE,
         NPF_INTERFACE,
         NPF_PORT,
         NPF_INTCONFIG  } NPF_DATATYPE
```

**Description**

All data types supported in the PDK.

## 2.2.3 NPF_RET

**Definition**

```
enum {  NPF_SUCCESS,
        NPF_OSAS_FAIL,
        NPF_INVALID_PARAMETERS,
        NPF_OUT_OF_MEMORY,
        NPF_DUPLICATE_CONTEXT,
        NPF_NO_INTERNAL_PORT,
        NPF_COMPONENT_UNINITIALIZED,
        NPF_COMPONENT_REINITIALIZED,
        NPF_NS_PATH_OVER_LENGTH,
        NPF_NS_NODE_IN_USE,
        NPF_NS_NODE_EXISTS,
        NPF_NS_NODE_NOTFOUND,
        NPF_CALLBACK_FAIL,
        NPF_FE_CONNECTION_FAIL,
        NPF_IF_INVALID_PORT_ASSOCIATION  } NPF_RET
```

**Description**

All return types supported in the PDK.

## 2.2.4 NPF_HANDLE

**Definition**

```
typedef uint32_t NPF_HANDLE
```

**Description**

The object ID of the operation destination, which is obtained from namespace API calls. A HANDLE may represent one or a group of forwarding plane objects, such as a virtual router, an FE, or an interface. The object handle must be obtained before API operations, and each operation can only pass in one HANDLE.

## 2.2.5 NPF_NSHDL

**Definition**

```
typedef uint32_t NPF_NSHDL
```

**Description**

The namespace provides the directory service to the other components to access basic system data, as well as more high-level logical data such as virtual router configuration. The namespace handle points to the node in the namespace.

## 2.2.6 NPF_CBHANDLE

**Definition**

```
typedef uint32_t NPF_CBHANDLE
```

**Description**

The APIs are grouped into different categories, and each API is associated with a callback handle. This handle is provided by the PDK when a user application registers for that category. The handle is returned to the user application, and is used by the application when making the same category of API calls. A handle obtained when registering with one category must not be used in another category of API calls. The user application does not interpret this callback handle.

## 2.2.7 NPF_USERCONTEXT

**Definition**

```
typedef void *NPF_USERCONTEXT
```

**Description**

Provided by the registering application at the time the application registers the APIs. Its value is not interpreted by the API components, but is put as a parameter when the PDK gets the response from forwarding plane and invokes a callback provided by the registering application.

The application need not enforce a unique user context for each registration. The PDK uses the callback handle of this user context to map the correct callback functions.

A single application-provided callback function may be registered in multiple situations. For example, an application may provide the same function in response to an administrative interface shutdown API and an abrupt interface shutdown event. In this case, the application registers both API callback and event callback with the same callback function, but a different user context. It is the application's responsibility to differentiate the callback type by checking the returned user context. The PDK does not interpret or modify the user context passed in by the application.

## 2.2.8 NPF_CORRELATOR

**Definition**

```
typedef void *NPF_CORRELATOR
```

**Description**

An application must provide an NPF_CORRELATOR for each invocation of the API call. This is required to support scenarios where an application might have multiple outstanding requests of the same type. For example, an application might have multiple function calls to assign IP addresses to different interfaces on the forwarding plane, all of which expect acknowledgements as asynchronous callbacks.

The NPF_ CORRELATOR provides a way for the application to differentiate the correlation of each asynchronous response to the same registered callback return. When the application makes API calls, it includes the NPF_ CORRELATOR in the request, and the API components do not interpret its value. When the PDK gets the response from the forwarding plane, it includes this same NPF_ CORRELATOR as well as the NPF_USERCONTEXT to this type of callback when invoking the registered callback function in response to this request.

## 2.2.9 NPF_VERBOSITY

**Definition**

```
enum NPF_VERBOSITY { NPF_ACK, NPF_NOACK, NPF_FAILACK }
```

**Description**

Defines the behavior of the asynchronous mode of operation present in the request. ACK causes the PDK to always send a response to the application indicating the operation results. NOACK indicates that no response should be generated. FAILACK indicates that a response should be sent only for unsuccessful results.

## 2.2.10 NPF_RESPONSE

**Definition**

```
typedef void *NPF_RESPONSE
```

**Description**

This defines the response type returned from the PDK APIs. It is passed in callback functions provided by applications. The PDK defines different data structures for different types of callback functions. An application must cast the void data into the correct structure types depending on the return callback type. There could be more than one response.

## 2.3 List

The CP-PDK provides a thread-safe double-link list that can convey any user-defined data structure. User applications must allocate memory for their data and de-allocate memory after deletion from the list. If the de-allocate function is provided, the list can also de-allocate the data in the list when the user deletes the list element.

There are two types of operations in the list. The first is the basic operation and the second is the iterator operation. Basic operations are multi-thread-safe. For iterator operations, always use an external lock while traversing the list.

## 2.4 Basic Operations

Basic operations are multiple thread-safe and can be invoked simultaneously.

### 2.4.1 npf_list_init

**Definition**

```
int npf_list_init(DList *, void (*destroy)(void *data));
```

**Description**

Initialize a list.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |
| `void (*destroy)` | The destroy function to de-allocate the user data in the list |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

### 2.4.2 npf_list_destroy

**Definition**

```
int npf_list_destroy(DList *);
```

**Description**

Remove all elements in the list and de-allocate user data.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

None.

**Return Values**

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.3 npf_list_size

**Signature**

```
int npf_list_size(DList *);
```

**Description**

Get the size of the list.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.4 npf_list_pushFront

**Signature**

```
int npf_list_pushFront(DList *, const void *);
```

**Description**

Push data to the front of the list.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |
| `void *` | The pointer to the new data |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.5 npf_list_pushBack

**Signature**

```
int npf_list_pushBack(DList *, const void *);
```

**Description**

Push data to the back of the list.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |
| `void *` | The pointer to the new data |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.6 npf_list_popFront

**Signature**

```
int npf_list_popFront(DList *, void **);
```

**Description**

Delete the first list element and return the pointer to the deleted user data.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

| | |
|---|---|
| `void *` | The pointer to the deleted data |

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.7 npf_list_popBack

**Signature**

```
int npf_list_popBack(DList *, void **);
```

**Description**

Delete the last list element and return the pointer to the deleted user data.

**Input Parameters**

| | |
|---|---|
| `DList *` | The pointer to the list |

**Output Parameters**

| | |
|---|---|
| `void *` | The pointer to the deleted data |

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.8 npf_list_popFrontFree

**Signature**

```
int npf_list_popFrontFree(DList *);
```

**Description**

Delete the first list element and de-allocate the user data if the destroy function is provided at list registration time.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.9 npf_list_popBackFree

**Signature**

```
int npf_list_popBackFree(DList *);
```

**Description**

Delete the last list element and de-allocate the user data if the destroy function is provided at list registration time.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.4.10 npf_list_isExist

**Signature**

```
int npf_list_isExist(DList *, const void *);
```

**Description**

Check if the pass-in pointer exists in the list.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |
| `void *` | The pointer to the pass-in data |

**Output Parameters**

None

**Return Values**

0

## 2.4.11 npf_list_getFirstData

**Signature**

```
void *npf_list_getFirstData(DList *);
```

**Description**

Get the first user data stored in the first list element.

**Input Parameters**

`DList *` The list pointer

**Output Parameters**

None.

**Return Values**

Pointer to the first user data.

## 2.4.12 npf_list_getLastData

**Signature**

```
void *npf_list_getLastData(DList *);
```

**Description**

Get the last user data stored in the last list element.

**Input Parameters**

`DList *` The list pointer

**Output Parameters**

None

**Return Values**

Pointer to the last user data.

## 2.4.13 Example of Basic Operations

Create a list, push data at the back, and then pop it out.

```
DList pipeline;
struct segment *seg;
void *data;
npf_list_init(&pipeline, free);
seg = (struct segment *)malloc(sizeof(struct segment));
npf_list_pushBack(&pipeline, seg);
...
npf_list_popFront(&pipeline, (void **)&data);
```

if (list->destroy != NULL) pipeline->destroy(data);

## 2.5 Iterator Operations

An iterator marks a position in the list. It provides a way to access a list element without exposing its underlying representation. When using iterator operations, it is important for user programs to ensure that the proper lock is acquired, since a series of iterator operations are usually called sequentially and there may be some dependency between them.

Iterators are also useful for inserting and deleting middle nodes. After inserting or deleting the nodes, the return iterator is different in other API calls. Refer to the following descriptions for details.

### 2.5.1 npf_list_itrCreate

**Signature**

```
DListItr *npf_list_itrCreate(DList *);
```

**Description**

Create an undefined iterator, return `NULL` if the list is empty.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

None.

**Return Values**

Pointer to an undefined iterator; `NULL` if the list is empty.

### 2.5.2 npf_list_itrDelete

**Signature**

```
int npf_list_itrDelete(DListItr *);
```

**Description**

Delete the iterator.

**Input Parameters**

| | |
|---|---|
| `DList *` | The list pointer |

**Output Parameters**

None.

**Return Values**

| | |
|---|---|
| `0` | Successful |
| `-1` | Not successful |

## 2.5.3 npf_list_itrFirst

**Signature**

```
int npf_list_itrFirst(DListItr *);
```

**Description**

Get the first iterator in the list.

**Input Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the current iterator |

**Output Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the first iterator of the list |

**Return Values**

0 if successful; -1 if the list is empty.

## 2.5.4 npf_list_itrLast

**Signature**

```
int npf_list_itrLast(DListItr *);
```

**Description**

Get the last iterator in the list.

**Input Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the current iterator |

**Output Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the last iterator of the list |

**Return Values**

0 if successful; -1 if the list is empty.

## 2.5.5 npf_list_itrNext;

**Signature**

```
int npf_list_itrNext(DListItr *);
```

**Description**

Get the next iterator of the pass-in current iterator.

**Input Parameters**

`DListItr *` The pointer to the current iterator

**Output Parameters**

`DListItr *` The pointer to the iterator following the pass-in iterator

**Return Values**

0 if successful; -1 if the pass-in iterator is undefined or the next is `NULL`.

## 2.5.6 npf_list_itrPrev

**Signature**

```
int npf_list_itrPrev(DListItr *);
```

**Description**

Get the previous iterator of the pass-in current iterator.

**Input Parameters**

`DListItr *` The pointer to the current iterator

**Output Parameters**

`DListItr *` The pointer to the iterator prior to the pass-in iterator

**Return Values**

0 if successful; -1 if the pass-in iterator is undefined or the previous iterator is `NULL`.

## 2.5.7 npf_list_itrInsNext

**Signature**

```
int npf_list_itrInsNext(DListiItr *, const void *);
```

**Description**

Insert new data next to the pass-in iterator

**Input Parameters**

`DListItr *` The pointer to the current iterator

**Output Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the iterator next to the new added one |
| `const void *` | The pointer to the newly inserted data |

**Return Values**

0 if successful, -1 if the pass-in iterator is undefined.

## 2.5.8 npf_list_itrInsPrev

**Signature**

```
int npf_list_itrInsPrev(DListItr *, const void *);
```

**Description**

Insert new data prior to the pass-in iterator.

**Input Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the current iterator |

**Output Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the iterator does not change. It remains the one next to the newly added iterator. |

**Return Values**

0 if successful, -1 if the pass-in iterator is undefined.

## 2.5.9 npf_list_itrRemove

**Signature**

```
int npf_list_itrRemove(DListItr *, void **);
```

**Description**

Delete the list element and return the pointer to the user data.

**Input Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the current iterator |

**Output Parameters**

| | |
|---|---|
| `DListItr *` | The pointer to the iterator next to the removed one, set to undefined if the removed one is the last one |
| `void *` | The pointer to the removed user data |

### Return Values

0 if successful; -1 if the pass-in iterator is undefined.

## 2.5.10 npf_list_itrRemoveFree

### Signature

```
int npf_list_itrRemoveFree(DListItr *);
```

### Description

Delete the list element and de-allocate the user data using a user-provided destroy function.

### Input Parameters

| | |
|---|---|
| `DListItr *` | The pointer to the current iterator |

### Output Parameters

| | |
|---|---|
| `DListItr *` | The pointer to the iterator next to the removed one, set to undefined if the removed iterator is the last one |

### Return Values

0 if successful; -1 if the pass-in iterator is undefined.

## 2.5.11 npf_list_itrGetData

### Signature

```
void *npf_list_itrGetData(DListItr *);
```

### Description

Return to the pointer inside the iterator that points to the user data.

### Input Parameters

| | |
|---|---|
| `DListItr *` | The pointer to the current iterator |

### Output Parameters

None.

### Return Values

Pointer to the user data of the current iterator; returns `NULL` if pass-in iterator is undefined.

## 2.6 Example of Iterator Operations

Traverse a list:

```
DListItr *itr;
void *data;
if ((itr = npf_list_itrCreate(&pipeline)) != NULL)
      return;
LOCK();
npf_list_itrFirst(itr);
do {
      data = npf_list_itrGetData(itr);
      ...
} while (npf_list_itrNext(itr) != -1);
UNLOCK();
npf_list_itrDelete(itr);
```

# *Part 3: Callback Mechanism*

# 3 Callback Mechanism

## 3.1 Callback Types

CP-PDK provides two different types of callback, which are described in the following sections.

### 3.1.1 Event Callbacks

Applications may have interest in events happening in the underlying hardware. This kind of callback is not in response to a particular API call, but is in response to an event that occurs. Applications must register for events of interest, and when these events happen, the PDK invokes the registered callback function. In this case, only the NPF_USERCONTEXT and result data are passed back to the application. Refer to the Configuration and Management API Reference Guide for a description of event callbacks.

### 3.1.2 Response to API Function Callbacks

Most of the APIs do not receive an immediate response, because they have to wait for a certain time to allow for the underlying hardware execution. These APIs return immediately without an execution result. When the result returns, the PDK invokes an asynchronous callback in another callback thread based on the verbosity level. In this case, both the original NPF_USERCONTEXT and NPF_CORRELATOR are passed back to the application along with the result data. An application must differentiate each callback based on the types NFP_USERCONTEXT and NPF_CORRELATOR.

## 3.2 Callback Interfaces

### 3.2.1 Event

#### 3.2.1.1 npf_xx_event_register

**Signature**

```
int npf_xx_event_register(   NPF_USERCONTEXT,
                             NPF_EVENT_CBFUNC,
                             NPF_CBHANDLE *   )
```

**Description**

Allows the application to register a callback function with the event related callback type, and associates a unique callback handle. The possible event categories are: fe, port, if, ipv4 and ipr.

**Input Parameters**

NPF_USERCONTEXT

NPF_EVENT_CBFUNC

**Output Parameters**

NPF_CBHANDLE

**Return Values**

NPF_SUCCESS
NPF_INVALID_PARAMETERS

**Async Callback**

None.

### 3.2.1.2 npf_xx_event_deregister

**Signature**

```
int npf_xx_event_deregister( NPF_CBHANDLE )
```

**Description**

Allows the application to de-register the event callback function associated with the unique previously assigned callback handle.

**Input Parameters**

NPF_CBHANDLE

**Output Parameters**

None.

**Return Values**

NPF_SUCCESS
NPF_INVALID_PARAMETERS

**Async Callback**

None.

### 3.2.1.3 NPF_EVENT_CBFUNC

**Definition**

```
typedef void (*NPF_EVENT_CBFUNC)(   NPF_USERCONTEXT,
                                    NPF_EVENT,
                                    NPF_HANDLE,
                                    NPF_RESPONSE   )
```

**Description**

The uniform event callback function format. The application names the content of NPF_RESPONSE based on the NPF_EVENT_CBTYPE. This function is invoked by the PDK whenever the registered event happens. Applications check the NPF_EVENT_CBTYPE to know what information is returned. NPF_USERCONTEXT and NPF_HANDLE are the original values passed in from the applications.

### 3.2.2 Event Callback Example

```
void onCB_FEEvent(  NPF_USERCONTEXT       context,
                    NPF_EVENT             event,
                    NPF_HANDLE            handle,
                    NPF_DATA              data  )
{
    switch (event) {
        case NPF_EVENT_FE_BIND:
            break;
        case NPF_EVENT_FE_UP:
            break;
        default:
    }
}
int main() {
    NPF_CBHANDLE cbhandle;
    NPF_EVENT_CBFUNC evencbfunc = &onCB_FEEvent;
    npf_fe_event_register((void*)getpid(), eventcbfunc, &cbhandle);
    ...
    npf_fe_event_deregister(cbhandle);
}
```

### 3.2.3 Responses to APIs

Most function calls in the CP-PDK APIs are asynchronous. They are grouped into different categories. An application must register a callback for a given category to get the callback response. At registration time, the PDK returns a callback handle. When an application needs to invoke the subsequent API calls from that category, the application must use the corresponding callback handle. An application can register or de-register a callback category at any time, or during initialization or during shutdown.

#### 3.2.3.1 npf_xxxx_register

**Signature**

```
int npf_xxxx_register(  NPF_USERCONTEXT,
                        NPF_xxxx_CBFUNC,
                        NPF_CBHANDLE *   )
```

**Description**

Allows the application to register a callback function with all the XXXX related callback types, and associates a unique callback handle.

**Input Parameters**

NPF_USERCONTEXT
NPF_XXXX_CBFUNC

**Output Parameters**
NPF_CBHANDLE

**Return Values**
NPF_SUCCESS
NPF_INVALID_PARAMETERS

**Async Callback**

None

### 3.2.3.2 npf_xxxx_deregister

**Signature**
```
int npf_xxxx_deregister( NPF_CBHANDLE )
```

**Description**

Allows the application to de-register the XXXX callback function that associates with the unique pre-assigned callback handle.

**Input Parameters**
NPF_CBHANDLE

**Output Parameters**

None.

**Return Values**
NPF_SUCCESS
NPF_INVALID_PARAMETERS

**Async Callback**

None.

### 3.2.3.3 NPF_XXXX_CBFUNC

**Definition**
```
typedef void (*NPF_XXXX_CBFUNC)(    NPF_USERCONTEXT,
                                    NPF_CORRELATOR,
                                    NPF_VERBOSITY,
                                    NPF_HANDLE,
                                    NPF_(___)_CBTYPE,
                                    NPF_RESPONSE   )
```

**Description**

The uniform XXXX callback function format. An application names the content of NPF_RESPONSE based on the NPF_XXXX_CBTYPE. The PDK invokes this function whenever the registered XXXX related event happens. Applications check the NPF_XXXX_CBTYPE to know the information that is

returned. NPF_USERCONTEXT, NPF_CORRELATOR, NPF_HANDLE and NPF_VERBOSITY are the original values passed in from the applications.

### 3.2.4 Asynchronous API Callback Example

```
void onCB_IF(     NPF_USERCONTEXT context,
                  NPF_CORRELATOR  correlator,
                  NPF_VERBOSITY   verbosity,
                  NPF_HANDLE      handle,
                  NPF_IP_CBTYPE   type,
                  NPF_DATA        data    )
{
    switch (type) {
    case NPF_IF_SET_IPADDR:
        break;
    case NPF_IF_SET_FORWARDING:
        break;
    default:
    }
}
int main() {
    NPF_CBHANDLE cbhandle;
    NPF_IF_CBFUNC ifcbfunc = &onCB_IF;
    npf_if_register("ConIF", ifcbfunc, &cbhandle);
    ...
    npf_if_set_ipaddr(fehandle, cbhandle, “Correlator”, NPF_ACK,
ipaddr);
    ...
    npf_if_deregister(cbhandle);

}
```

# *Part 4: Memory Allocation and State Maintenance*

# 4 Memory Allocation and State Maintenance

The memory allocation and usage model for the API implementation is as follows:

The entity that may be an application or API implementation that allocates memory, is responsible for freeing it.

When an application makes a CP-PDK API invocation and passes in any data for which it allocated the memory, the API implementation always copies that data and returns a `HANDLE`. The PDK also guarantees that the memory passed in is used only for the duration of the API call. The application can use it in any manner desired, once the API call is complete and control is returned to the application. An application can only access the PDK data through the `HANDLE` that was returned from the PDK.

Similarly, when the API implementation invokes a registered application callback, the application must ensure that it copies the passed data, if required, after the duration of the callback function execution.

There are four types of transient data in the PDK:

- Data created by the application, such as routing information or FTN/ILM information
- Data collected from the FEs that cannot be changed, such as MAC addresses
- Data collected from the FEs that can be changed, such as MTU
- Traffic statistics in the FEs

The PDK will cache type 2 and type 3 data in the Configuration and Management module, since it is not as volatile as type 1 and type 4. It is under the PDK's control. For routing information and MPLS FIB information, the tables are stored in the user application. The PDK does not keep a state of that data except overhead information generated by the PDK, such as inter-FE forwarding labels.

# *Part 5: Logging Service*

# 5 Logging Service

The PDK provides a basic logging service for both user programs and internal PDK components to record runtime messages. Messages can be logged by component or based on severity.

## 5.1 Requirements

The following are requirements of the logging service:

- Write results to a specific log file
- Timestamp of every entry
- Show name of source file and function in use
- Display variables in dynamic format
- Granulate the severity of the debug message
- Maintain the log file within a certain length
- Log by severity and/or component, as assigned from command line

## 5.2 Data Types

### 5.2.1 npf_logger_verbosity

**Definition**

```
enum {      LOG_VERB_COMP,
            LOG_VERB_LEVEL,
            LOG_VERB_FUNC,
            LOG_VERB_LINE,
            LOG_VERB_FILE,
            LOG_VERB_PID,
            LOG_VERB_TIME  }
```

**Description**

The verbosity levels provided by the PDK.

## 5.2.2 npf_logger_level

**Definition**

```
enum {      LOG_DISABLED,     // logging is disabled
            LOG_TRACE,        // trace messages, function
                                 entered/exited
            LOG_INFO,         // information message, print debugging
            LOG_ERR,          // error condition
            LOG_CRIT,         // critical error, unrecoverable
      } npf_logger_level
```

**Description**

The debug levels provided by the PDK.

## 5.2.3 npf_logger_component

**Definition**

```
enum {   LOG_PDKMGR,    // pdk manager
         LOG_NS,        // namespace
         LOG_LBM,       // label manager
         LOG_TPM,       // topology manager
         LOG_CM,        // config manager
         LOG_BD,        // binding discovery
         LOG_IPV4,      // ipv4
         LOG_INTSERV,   // intserv
         LOG_DS,        // diffserv
         LOG_MPLS,      // mpls
         LOG_ATM,       // ATM
         LOG_TOPO,      // topology
         LOG_CB,        // callback
         LOG_VIC,       // virtual device
         LOG_PH,        // packet handler
         LOG_APP,       // application
         LOG_TPI,       // transport plugin
         LOG_FP_QOS,    // FP QoSManager
         LOG_FP_MPLS    // FP MPLS Manager      }
```

**Description**

The debug components supported by the PDK.

## 5.3 Interfaces

### 5.3.1 npf_logger_Start

**Signature**

```
int npf_logger_Start ( char *filename )
```

**Description**

Open the log file and start the logging service.

**Input Parameters**

| | |
|---|---|
| `filename` | The log filename and path |

**Output Parameters**

None.

**Return Values**

0 if success; otherwise return –1.

### 5.3.2 npf_logger_Stop

**Signature**

```
int npf_logger_Stop (void)
```

**Description**

Close the log file and turn off the logging service.

**Input Parameters**

None.

**Output Parameters**

None.

**Return Values**

0 if success.

### 5.3.3 npf_logger_SetLevel

**Signature**

```
int npf_logger_SetLevel ( npf_logger_component, uint32_t
npf_logger_level )
```

**Description**

Set the component and level of the logging service.

**Input Parameters**

| | |
|---|---|
| `npf_logger_component` | The component for which the level is set |
| `npf_logger_component` | The severity level |

**Output Parameters**

None.

**Return Values**

0 if success.

## 5.3.4 npf_logger_SetVerbosity

**Signature**

```
int npf_logger_SetVerbosity( npf_logger_verbosity )
```

**Description**

Set the log verbosity.

**Input Parameters**

| | |
|---|---|
| `npf_logger_verbosity` | The logger verbosity |

**Output Parameters**

None.

**Return Values**

0 if success.

## 5.3.5 npf_logger_Write

**Signature**

```
int npf_logger_write (  npf_logger_component,
                        npf_logger_level,
                        format,
                        args ...    )
```

**Description**

Log the message to the log file.

**Input Parameters**

| | |
|---|---|
| `npf_logger_component` | The component to which this message refers |
| `npf_logger_level` | The severity level |
| `format` | The format of the message |
| `args` | The argument(s) |

**Output Parameters**

None.

**Return Values**

`0` if success.

## 5.4 Logging Examples

The following are examples of the logging service.

```
npf_logger_Start(“/tmp/log.txt”);
npf_logger_SetLevel(LOG_CM, LOG_ERR);

npf_logger_SetVerbosity(LOG_TIME);
npf_logger_Write("Connection Refuse!");
npf_logger_Write("Callback %dth from %s!", 100, "FP");
npf_logger_Write("This will not show up!");
npf_logger_Stop();
```

Logged messages in `/tmp/log.txt`:

```
(pid|file:function:line|date time|component:level - message)
22314|log.c:main:85|05/04 12:57:53.225|1:3 - Connection Refused!
22314|log.c:main:86|05/04 12:57:53.226|0:6 - Callback 100th from FP!
```

# Part 6: Locks and Multiple Threads

# 6 Locks and Multiple Threads

## 6.1 Locks

Locks are used in the PDK to protect system data for thread safety. There are two levels of operations inside the PDK that should be guarded by locks:

1. For intra-component operations
2. For inter-component operations

The intra-domain operations need to only access the data inside a single component. The implementations of the APIs exposed by these components are mostly intra-component. Each component should use internal locks to protect its own data.

For inter-component operations, each component should expose a per-component lock. These locks are coordinated in the inter-component operations so that deadlock is avoided while thread safety is provided. This single lock semantic is exposed by a component, while internal to the implementation of this single lock, multiple mutexes or locks can be used in a coordinated manner to enhance the efficiency of the locking mechanism.

There are two recommendations for the locks used with the PDK:

1. All locks should be read-write, which promotes a greater degree of concurrency and efficiency while enforcing protection.
2. An order of nesting of the locks is used to avoid deadlock among the per-component locks for inter-component operations. This order says that a lower lock in the order must be unlocked before a higher order lock is locked. The order of the locks, from high to low, is:
    1. PDK manager
    2. Configuration manager
    3. Namespace
    4. Binding and discovery manager
    5. Topology manager
    6. Label manager
    7. IPV4 manager
    8. MPLS manager
    9. IntServ manager
    10. DiffServ manager
    11. ATM manager

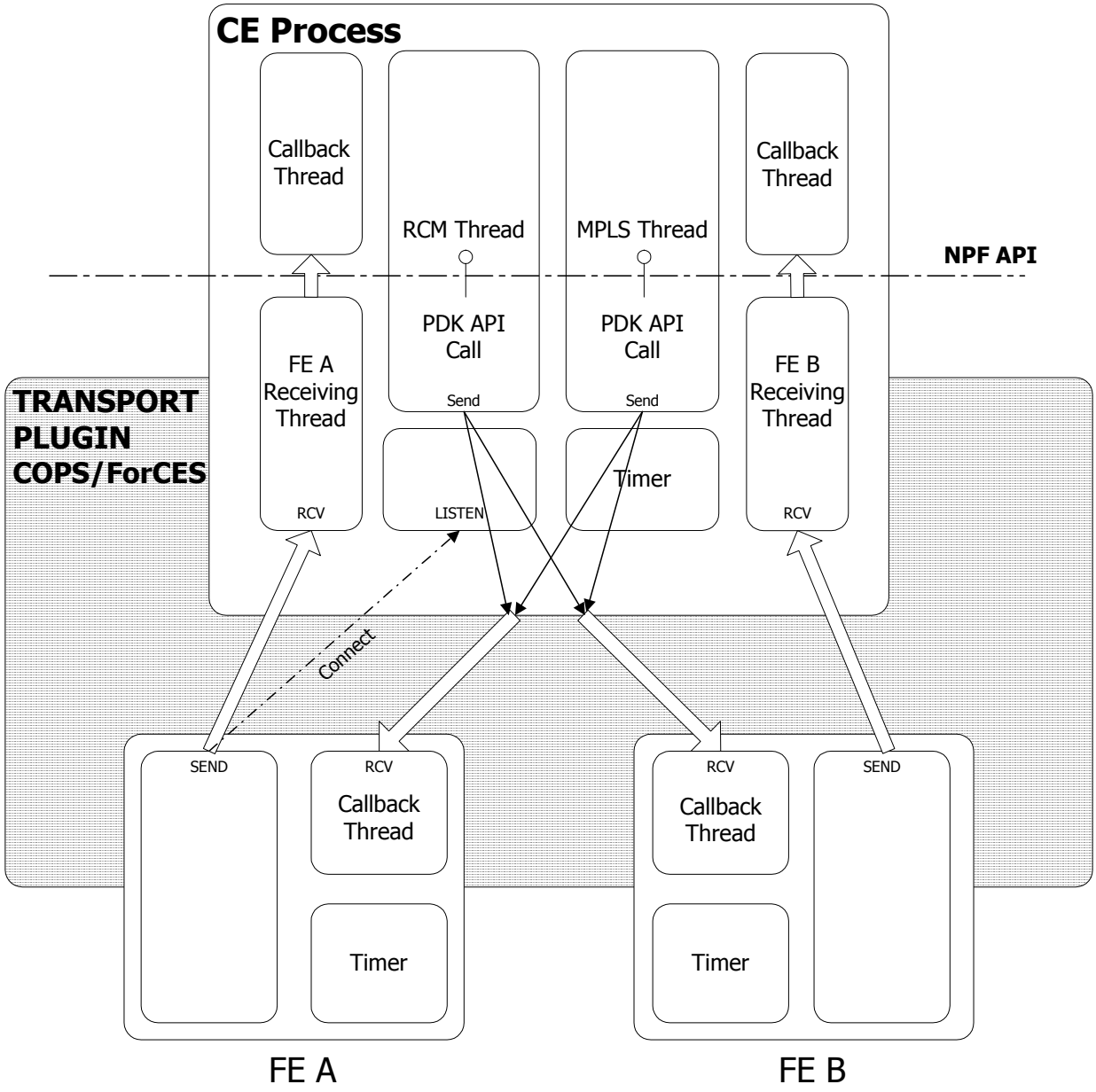
## 6.2 Single Process and Multiple Threads

In this release of the PDK, user applications and PDK components are compiled into one single executable file. Each user program has its own thread to invoke PDK API calls. Each FE has one thread in the CE to receive the message coming from that FE. The main function of that thread is to receive the message from the FE and execute the callback based on the incoming message.

When an asynchronous API is invoked in the application threads, it returns immediately without having a result. After the underlying FE finishes the command, the receiving thread in CE is notified and triggers a series of callback to return to the application. Besides the application threads and receiving threads, there is a timer thread and connection listen thread that are parts of COPS.

Figure 2 illustrates the multiple threads in one process PDK implementation.

**PDK Under One Process**




* Each rectangle represents a single thread
** Each FE uses two threads but a single socket descriptor to send and receive message between CE
*** Thread number in CE = 1 listen + 1 Timer + X FE Callbacks + Y applications + Z callback threads

**Figure 2: PDK under multiple threads**

A thread pool that contains multiple threads executes the application callback. When the receiving thread FP Plugin in the control plane receives the response, it executes all the internal callbacks in its context and pipes the response information to one of the callback threads in the thread pool to have the callback thread run the application callback jobs. In this way, the PDK does not run the risk that the application callback never returns.

It is important to protect the global data because of the multiple thread issues. Data can be modified only by a component that owns the data. The component may use a per-component lock to protect all the internal data or use a per-data lock to minimize the lock time. This should be decided case-by-case, depending on the situation.

# Part 7: Initialization and Shutdown

# 7 Initialization and Shutdown

## 7.1 Initialization

Initialization starts from a PDK application, such as the configuration application. The application first calls `PDKinit()` exposing the PDK manager. The PDK manager initializes the internal PDK components one-by-one in a well-defined order. This order must take two considerations into account:

- The Control Plane portion of the PDK should configure its internal components before accepting any binding from forwarding elements. In this way, when an FE comes to bind, the CE portion of the PDK is ready to control it. This implies that the part of the FP Plugin that first accepts a binding request from an FE should be the last one to be initialized.
- A PDK component may need to register callback functions with the FP Plugin for completing routines of CE to FE calls and event notifications from FE to CE. This means that the API portion of the FP Plugin should be initialized before those components that register callbacks.

These considerations require breaking the FP Plugin into two parts: the API part and the FE connection part. These two parts will be initialized separately in the following order:

- FP Plugin API
- Configuration manager
- Namespace
- Binding and discovery manager
- Topology manager
- Label manager
- IPV4 manager
- MPLS manager

After initialization, the configuration application reads the initial configuration information from the persistent IDB, and uses it to populate the namespace and configure all other PDK components. After initialization, the PDK Manager calls `PDKStart()` to enable the connection part of the FP Plugin.

## 7.2 Shutdown

Shutdown is triggered by a PDK application, such as the configuration application. The configuration application reads the current configuration state from each PDK component and saves this information into the IDB. It then calls the PDK manager to shut down the PDK. The PDK manager calls the configuration manager to shut down each component one-by-one in the defined order.

Shutdown can also be triggered by an `FE Unbind` event. When an `FE Unbind` event is sent back to CE, the configuration manager calls all callback functions registered for `FE DOWN` event for this FE.

Each component in the PDK must provide a function for an FE down event, `onCB_FEDOWN()`. The `onCB_FEDOWN()`calls back all the outstanding requests, closes all the open nodes, and releases all locks. The PDK Manager calls the administrative `SHUTDOWN()`, and the Transport Plugin calls the `FEDown` event callback, but they do the same thing to clean the internal data structures.

# *Part 8: Naming Guidelines*

# 8 Naming Guidelines

## 8.1 External API Function names

Function names comprise three parts:

1. The `npf_` preface
2. The API name
3. The name of the function

The name portion of the function should start with a capitalized letter and should use additional capital letters for separating additional words. If an acronym is included in the name, the acronym should be typed as it normally would, such as VoIP or COPS. Function names should always contain a verb.

The following are some examples:

```
npf_ipv4_AddRoute
npf_cm_GetL3Properties
npf_ipv4_RegisterARPCallback
npf_ipv4_DeregisterARPCallback
```

***Note:*** For the register and deregister functions, `Register` and `Deregister` must come before the event/callback type.

The abbreviated API names to include in the function name prefix are:

| | |
|---|---|
| `IPv4` | `IPv4` |
| `IPv6` | `IPv6` |
| `IPv6 Tunnel` | `IPv6TM` |
| `Configuration and Management` | `cm` |
| `Namespace` | `ns` |
| `MPLS` | `MPLS` |
| `IntServ` | `INTSERV` |
| `DiffServ` | `DS` |
| `ATM` | `ATM` |

## 8.2 Internal PDK Function Names

For functions internal to the PDK implementation but used as an interface between components (for example, if CM exposes a function to other internal components but does not expose this function to external applications), the function name should have the component name, an underscore, and then the function name. The name should start with a capitalized letter and should use additional capital letters for separating additional words. If an acronym is included in the name, the acronym should be typed as it normally would, such as VoIP or COPS. Function names should always contain a verb.

The following are some examples:

```
topo_LookupTopology
lblman_AllocateLabel
```

The abbreviated API names to include in the function name prefix are:

| | |
|---|---|
| IPv4 | ipv4 |
| Configuration and Management | cm |
| Namespace | ns |
| MPLS | mpls |
| IntServ | intserv |
| DiffServ | ds |
| Topology Manager | topo |
| Label Manager | lblman |
| ATM | atm |

## 8.3 Variable Names

Variable names include function parameter names, structure/union members, and any other variables defined by an API. They must be in lower case with an underscore ( _ ) used as a separator.

The following are some examples:

```
route_entry_list
port_number
```

## 8.4 Names for API Level Types

These naming guidelines are only for types exposed at the NPF API level. Types defined in the PDK implementation should use the naming guidelines found elsewhere in this document.

Type names include structure/union types, enums and typedefs. These should be prefaced with `npf_` and the first letter of the type name should be lower case. The name should be terminated with `_t` to indicate that it is a type. If the type name is composed of multiple words, additional words should be started with an upper case letter. Acronyms should be typed as they normally would, such as VoIP or COPS. This is a possible exception to the first letter lower case rule.

The following are some examples:

```
npf_routeEntry_t
npf_IPAddress_t
```

## 8.5 Constants

Any numerical should be a constant. This includes preprocessor constants such as `#define`, as well as the values inside an enumeration. These numerical constants should be in all caps and use underscores (_) as separators between multiple words.

The following is an example:

IPV4_MAX_ROUTES