Configuration and Management API

Reference Guide

Control Plane-Platform Development Kit 2.11

March 2004

Contents

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: Configuration and Management

1 Configuration and Management

Configuration and Management (CnM) exposes a mechanism to receive events about the system comprising Forwarding Element (FE), ports and interfaces. It also provides a synchronous API to query capabilities of FEs in terms of status, ports, attributes, and so on.

This document describes the Configuration and Management API of the Control Plane Platform Development Kit (CP-PDK). The CnM API is intended to provide a uniform API to primarily deal with events related to forwarding elements, ports and interfaces.

1.1 Configuration and Management API

The CP-PDK currently includes the following sets of Configuration and Management APIs:

- Event management: Allows applications to register and receive notifications when certain events (such as an FE/port status changes (up/ down)) occur in the underlying forwarding plane.
- FE management: Allows applications to query overall FE capabilities. This function is called by administrative or management applications to know about FE.

1.2 Terminology

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)
CnM	Configuration and Management

1.3 Event Management

The Event Management API allows applications to register for events that can occur in the underlying forwarding plane. For example, an application may want to register callbacks for receiving notification when an FE comes up, a port goes down on an FE, an FE goes down, and so on. This API allows applications using the CP-PDK APIs to register callbacks for such generic events. The reported events are sub-divided into the following categories:

1. FE events	`NPF_EVENT_FE_BIND`	FE bind has occurred
	`NPF_EVENT_FE_UP`	FE has come up
	`NPF_EVENT_FE_DOWN`	FE has gone down
2. Interface events	`NPF_EVENT_IF_UP`	If the interface is up
	`NPF_EVENT_IF_DOWN`	If the interface is down
	`NPF_EVENT_IF_IPADDR_CHANGE`	If the IP address has changed
	`NPF_EVENT_IF_DELETE`	If the interface is deleted
	`NPF_EVENT_IF_IPFORWARDING`	If the IP forwarding status has changed
3. Port events	`NPF_EVENT_PORT_UP`	Port has come up
	`NPF_EVENT_PORT_DOWN`	Port has gone down
	`NPF_EVENT_PORT_MTU_CHANGE`	Port's MTU size has changed
	`NPF_EVENT_PORT_LINKSPEED_CHANGE`	Port's LinkSpeed has changed

The Event Management API allows applications to register callbacks for any of the above categories of events that are invoked when the corresponding events occur on the forwarding plane.

1.4 Resolving Events and Callbacks

The API implementation must guarantee the following behavior with respect to events and callbacks:

If multiple applications register for an event, and the event can occur as the result of an API callback invoked by any of the applications, the call completion result (callback) must reach only the application that made the API call, and all other applications must be notified of the event.

For example, if P1 and P2 (applications/processes, and so on) register for an `NPF_EVENT_IF_DOWN` event for a particular interface, and P1 makes a Configuration and Management API call to shut down the same interface. In this scenario, when the FE shuts down the interface, the API implementation invokes the corresponding callback registered by P1 to indicate the status of the API call.

Additionally, since both P1 and P2 had also registered for the event, the callbacks they had registered for the `NPF_EVENT_IF_DOWN` event are also invoked. P1 must resolve the fact that it has received

both an API call completion and an event callback for the same event that occurred on the forwarding plane, and possibly ignore the event.

The API implementation should not assume that all events at the CP-PDK API level are exactly matched by events on the forwarding plane level.

For example, `NPF_EVENT_IF_IPADDR_CHANGE` (change in the IP address of an interface) is an event at the PDK API level, since an application might be interested in being notified when the IP address of an interface changes. The IP address of an interface is changed by an application invoking the corresponding Configuration and Management API call. When the forwarding element actually changes the IP address as desired, the call completion status is reported through a corresponding callback *only* to the application that invoked the API call to change the IP address. There is, however, no corresponding asynchronous event from the forwarding element that the IP address of an interface has been changed.

Accordingly, a corresponding event must also be generated for applications that need to be notified. The API implementation must resolve such scenarios where call completions must generate corresponding events to external applications.

The following events are currently supported:

1. `NPF_EVENT_FE_BIND`: Indicates to an application that an FE is now bound to the CE. This event results when the CE has just discovered a new FE. At this time, the FE has not been configured: its ports do not have IP addresses assigned and it does not have a routing table, and so on. Only a configuration application must register for this event, on which it uses the Configuration and Management API functions to configure the FE and its properties. Applications must register for this event in order to configure the new forwarding element and for supporting hot swap ability of forwarding planes.
2. `NPF_EVENT_FE_UP`: Generated after configuration application completes all the requests for this FE bind event.
3. `NPF_EVENT_FE_DOWN`: Generated if an FE goes down. An additional parameter in the callback indicates the reason for the event: administrative shutdown, malfunctioning of the FE, and so on.
4. `NPF_EVENT_IF_UP`: Indicates that an interface on an FE has come up.
5. `NPF_EVENT_IF_DOWN`: Indicates that an interface on an FE has gone down.
6. `NPF_EVENT_IF_DELETE`: This event is generated when a FE goes down for all interfaces that have configured on the FE.
7. `NPF_EVENT_IF_IPADDR_CHANGE`: Indicates that the interface's IP address has been changed.
8. `NPF_EVENT_IF_IPFORWARD_CHANGE`: Indicates that the interface's IP forwarding status has been changed.
9. `NPF_EVENT_PORT_MTU_CHANGE`: Indicates that the port's MTU has been changed.
10. `NPF_EVENT_PORT_LINKUP`: Indicates that the port's link went up.
11. `NPF_EVENT_PORT_LINKDOWN`: Indicates that the port's link went down.
12. `NPF_EVENT_PORT_LINKSPEED_CHANGE`: Indicates that the port's line speed has been changed.

1.5 Callback for Event Notification

Syntax

```
typedef void (*NPF_EVENT_CBFUNC)(
       IN: NPF_USERCONTEXT    context,
       IN: NPF_HANDLE         obj_handle,
       IN: NPF_EVENT          event,
       IN: NPF_DATA*          response );
```

Description

The callback function must be registered for any category of events in which an application is interested. The API implementation invokes the correct callback when any event in the specified category occurs.

Input Parameters

`context`	Registration context, known and interpreted by the application
`obj_handle`	The `NPF_HANDLE` to the object on which the event occurred. For example, if the event was `NPF_FE_DOWN`, `obj_handle` is the `NPF_HANDLE` to the FE object that went down.
`event`	One of the events is listed in Section 1.4, Resolving Events and Callbacks
`response`	Well-defined data structure that contains relevant information about the event that occurred

1.6 Register Callback for FE Event Notification

Syntax

```
NPF_RET npf_fe_event_register (
                     IN: NPF_USERCONTEXT    context,
                     IN: NPF_EVENT_CBFUNC   cb_fn,
                     OUT: NPF_CBHANDLE      fe_evnt_cbhandle );
```

Description

The registration function for registering a callback for FE events in which an application is interested. The API implementation invokes the callback function on occurrence of any event in the specified category.

Input Parameters

`context`	Registration context, known and interpreted by the application
`cb_fn`	Pointer to the callback function of the prototype defined in the previous section. The events in this category are: NPF_EVENT_FE_BIND NPF_EVENT_FE_UP NPF_EVENT_FE_DOWN

Output Parameters

`fe_evnt_cbhandle`	Unique callback registration handle

1.7 Deregister Callback for FE Event Notification

Syntax

```
NPF_RET npf_fe_event_deregister (
         IN: NPF_CBHANDLE cb_handle );
```

Description

De-registers a callback from any FE events.

Input Parameters

`cb_handle`	Callback handle, obtained during registration

1.8 Response Data for FE Event Callbacks

The last parameter in the callback function described in Section 1.5, Callback for Event Notification, `NPF_DATA`, contains a pointer to event-specific data. Although it is a void pointer, it points to a well-defined structure, and the application handling the callback must ensure that it casts the pointer to the correct type before using it. The pointer is passed as a generic void pointer, so the same function signature can be used for passing event-specific response data. Thus, response data for a different event could be a different structure. Different structures used for each category or case is explained below:

a. `NPF_EVENT_FE_BIND & NPF_EVENT_FE_UP`

In this case, a new FE has bound to the system and the application requesting to be notified might be interested in downloading new configuration information for the new FE. As mentioned earlier, the applications need not register for this event, if the FE was previously bound to the CE, before the last reboot.

The administrative or management applications that desire support for hot swap ability must register for this event. In this case, the FE configuration information is not known to the API implementation. The structure passed to the application for this event contains details of the new FE, such as basic system information, capabilities, number of ports, and so on, as indicated by the following:

```
typedef struct NPF_FE_ATTRIBS_tag
{
   char       desc[256];       /* FE description */
   char       name[128];       /* unique name of FE */
   time_t     stamp;           /* time stamp */
   uint32_t   id;              /* FE ID for this FE */
   int        num_ports;       /* number of ports on this FE */
   int        status;          /* UP, DOWN */
   HWADDR     hwaddr;          /* MAC address, possibly */
   IPADDR     ipaddr;          /* IP address, if any */
} NPF_FE_ATTRIBS;

typedef struct NPF_FE_tag
{
   uint32_t          id;      /* FE ID for this FE, = attribs.id */
   NPF_FE_ATTRIBS    attribs;
   NPF_FE_CAPS       caps;
} NPF_FE_t;
```

b. `NPF_EVENT_FE_DOWN`

The response data in this case contains the reason for the FE going down. For example, it could contain a detailed description of the problem that occurred for diagnostic purposes.

1.9 Register Callback for Interface Event Notification

Syntax

```
NPF_RET npf_if_event_register ( IN: NPF_USERCONTEXT   context,
                  IN: NPF_EVENT_CBFUNC   cb_fn,
                  OUT: NPF_CBHANDLE      if_evnt_cbhandle );
```

Description

The registration function for registering a callback for the interface events in which an application is interested. The API implementation invokes the callback function on occurrence of any event in the specified category.

Input Parameters

`context`	Registration context, known and interpreted by the application
`cb_fn`	Pointer to an event callback function as defined in Section 2.2 The events in this category are: NPF_EVENT_IF_UP NPF_EVENT_IF_DOWN NPF_EVENT_IF_DELETE NPF_EVENT_IF_IPADDR_CHANGE NPF_EVENT_IF_IPFORWARDING_CHANGE

Output Parameters

`if_evnt_cbhandle`	An unique callback registration handle

1.10 Deregister Callback for Interface Event Notification

Syntax

```
NPF_RET npf_if event_deregister (IN: NPF_CBHANDLE cb_handle );
```

Description

Deregisters a callback from interface events.

Input Parameters

`cb_handle`	Callback handle, obtained during the callback registration

1.11 Response Data for Interface Event Callbacks

The last parameter in the callback function described in Section 1.5, Callback for Event Notification, `NPF_DATA`, contains a pointer to the event-specific data. Although it is a void pointer, it points to a well-defined structure, and the application handling the callback must ensure that it casts the pointer to the correct type before using it. The pointer is passed as a generic void pointer, so the same function signature can be used for passing event-specific response data. Thus, response data for each different event could be a different structure.

Different structures used for each category or case are explained below:

`NPF_EVENT_IF_UP`
`NPF_EVENT_IF_DOWN`
`NPF_EVENT_IF_DELETE`
`NPF_EVENT_IF_IPADDR_CHANGE`
`NPF_EVENT_IF_IPFORWARD_CHANGE`

The response data in all these cases contains the interface attributes in a structure NPF_INTERFACE_t as defined by the following:

```
typedef struct NPF_IF_IP_ATTRIBS_tag
{
   IPA      ipaddr;
   IPA      submask;
   Int      ipfwding;   /* ENABLED, DISABLED */
} NPF_IF_IP_ATTRIBS;

typedef struct NPF_interface_tag
{
   char              name[128];
   char              desc[256];
   int               status;        /* UP, DOWN, TESTING */
   int               index;         /* Interface index */
   char              vifname[128];  /* virtual interface name */
   NPF_IF_IP_ATTRIBS ipattribs;
} NPF_INTERFACE_t;
```

1.12 Register Callback for Port Event Notification

Syntax

```
NPF_RET npf_port_event_register (IN: NPF_USERCONTEXT  context,
          IN: NPF_EVENT_CBFUNC     cb_fn,
          OUT: NPF_CBHANDLE        port_evnt_cbhandle);
```

Description

The registration function for registering a callback of any port events in which an application is interested. The API implementation invokes the callback function on occurrence of any event in the specified category.

Input Parameters

context	Registration context known and interpreted by the application
cb_fn	Pointer to an event callback function as defined in Section 2.2. Currently defined events in this category are: NPF_EVENT_PORT_LINKUP NPF_EVENT_PORT_LINKDOWN NPF_EVENT_MTU_CHANGE NPF_EVENT_LINKSPEED_CHANGE

Output Parameters

port_evnt_cbhandle	Unique callback registration handle

1.13 Deregister Callback for Port Event Notification

Syntax

```
NPF_RET npf_port_event_deregister (IN: NPF_CBHANDLE cb_handle );
```

Description

Deregisters a callback from port events.

Input Parameters

`cb_handle`	Callback handle obtained during the callback registration

1.14 Response Data for Port Event Callbacks

The last parameter in the callback function described in Section 1.5, Callback for Event Notification, `NPF_DATA`, contains a pointer to event-specific data. Although it is a void pointer, it points to a well-defined structure, and the application handling the callback must ensure that it casts the pointer to the correct type before using it. As the pointer is passed as a generic void pointer, the same function signature can be used for passing event-specific response data. Thus, response data for each different event could be a different structure. Different structures used for each category or case is explained below:

`NPF_EVENT_PORT_LINKUP`
`NPF_EVENT_PORT_LINKDOWN`
`NPF_EVENT_PORT_MTU_CHANGE`
`NPF_EVENT_PORT_LINKSPEED_CHANGE`

The response data in all these cases contains the port attributes in a structure `NPF_PORT_t` as defined by the following:

```
typedef struct NPF_PORT_ATTRIBS_tag
{
NPF_PORT_TYPE     type; /* ETHERNET, ATM, etc. */
int   status;     /* admin control purpose */
int   linkstatus; /* FE reports it */
uint32_t    id;   /* PortID */
HWADDR      macaddr;
Int   minTxRate;
int   maxTxRate;
int   curTxRate;
int   MTU;
int   internal;   /* if it is internal port or not */
} NPF_PORT_ATTRIBS;
```

1.15 FE Management

This API supports the overall management of the forwarding plane, and is used by applications that are aware of the fact that the underlying forwarding plane could be composed of multiple forwarding elements. The API allows applications to obtain overall properties of, initialize, and shut down a forwarding element. The administrative or configuration applications can also initialize the properties of FE during CE/FE initialization.

1.16 Get FE Capabilities

Syntax

```
NPF_RET npf_fe_get_capabilities(
        IN:  uint32_t               feId,
        OUT: NPF_FeCapability_t     **feCaps);
```

Description

This is a synchronous nature of call where the capabilities of the specified FE are received when a `feId` is given. The capabilities returned are described in Section 1.8, Response Data for FE Event Callbacks. For a successful function call, the buffer returned has to be freed by the caller application. For unsuccessful calls the `feCaps` pointer returned is `NULL`.

The structure `NPF_FeCapability_t` is defined as:

```
typedef     struct      NPF_FeCapability
{
   uint32_t feId;       /* FE Identifier */
   time_t   timeStamp;  /* Time Stamp */
   IPADDR   feIpAddr;   /* IP Address of FE blade */
   HWADDR   feMacAddr;  /* MAC Address of FE blade */
   Int      status;     /* Status (Up/Down) of FE */
   uint32_t numPorts;   /* Number of Ports present in FE */
   NPF_PORT_ATTRIBS  *ports;  /* Port Attributes of numPorts ports */
} NPF_FeCapability_t;
```

In this struct `ports` is also a pointer. So for successful calls if `numPorts` member in struct defined above is non-zero then `feCaps->ports` has to be freed first and then `fePorts` pointer has to be freed.

Input Parameters

`feId`	Forwarding Element Identifier
`feCaps`	Pointer to Pointer to FE Capabilities

Return Values

`NPF_SUCCESS`	API call has been successfully made
`NPF_FAILURE`	API Call has failed