Forwarding Plane Module

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Revision History

Revision	Description	Date	Author
2.11	Updated for release 2.11	March 2004	Shabbir Ali
2.1	Updated for release 2.1	December 2003	Shabbir Ali
2.0	Updated for Release 2.0	August 2003	Shabbir Ali

Part 1: Introduction

1 Introduction

This document provides information on the forwarding plane (FP) module of the Control Plane Platform Development Kit (CP-PDK) that resides on the data plane. The FP module utilizes a FP Application Program Interface (API), such as, the IXA SDK 3.51 for IXP2400/2800, for interfacing with the underlying Network Processing Unit (NPU). The FP module receives the National Processing Forum (NPF) API invocations from the backend API of the CP-PDK transport plug-in and maps them to the corresponding FP API invocations. The FP module is also responsible for binding and reporting capabilities of the forwarding plane to the control plane.

The FP module, with this release of the CP-PDK, is targeted for VxWorks* and Linux* operating systems. In VxWorks based data plane, the FP module invokes the forwarding plane API (FPAPI) directly, such as, IXA SDK 3.51. For Linux-based data plane, as the FPAPI can reside in the kernel mode, an additional control messaging layer, exposing ioctl based APIs, exposes the FPAPI. But the overall design of the FP module remains the same.

The current FP module is designed to work with the IXA SDK 3.51 as the FPAPI. The FP module consists of a portable layer and an NPU platform-specific layer. The portable layer is independent of the FPAPI.

1.1 Terminology

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

Table 1. Terminology table

Terms	Expansion
ARP	Address Resolution Protocol
CC	Core Component Packet Processing Entity
Control Element (CE), Control Plane (CP)	In a separated control/data system, refers to the processor(s) responsible for control and configuration of forwarding elements. Used interchangeable with Control Plane (CP)
COPS	Common Open Policy Service Protocol
CORBA	Common Object Request Broker Architecture (http:www.omg.org)
CP-PDK	Control Plane Platform Development Kit
EE	Execution Engine
FEC	Forward Equivalence Class
FIB	Forward Information Base
ForCES	Forwarding and Control Element Separation protocol, currently being standardized at IETF
Forwarding Element (FE), Forwarding Plane (FP)	In a separated control/data system, refers to the processor(s) responsible for fast path forwarding of data. Used interchangeably with FP.
ICMP	Internet Control Message Protocol

Intel© XScale™ core	Forms the core of the IXP 2400 and 2800
IXA	Internet eXchange Architecture
IXP 1200, IXP 2400/2800	Network processors in Intel's IXA family.
L2T	Layer 2 Table maintained by the IXA SDK 3.51
MPLS	Multiprotocol Label Switching
NHLFE	Next Hop Label Forwarding Entry
NHT	Next Hop Table maintained by the IXA SDK 3.51
NPF	Network Processing Forum
OSPF	Open Shortest Path First (routing protocol)
PT	Prefix Table maintained by the IXA SDK 3.51
RIP	Routing Information Protocol

1.2 References

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

Table 2. Reference table

Reference	Description
[1]	CP-PDK Software Architecture Overview
[2]	CP-PDK Configuration and Management API Reference
[3]	CP-PDK Forwarding Plane Plug-in API Reference
[4]	CP-PDK IPv4 API Reference
[5]	CP-PDK Differentiated Services API Reference
[6]	CP-PDK Co-located Transport Plug-in Design Reference
[7]	CP-PDK Protocol Support Services Design Reference
[8]	An Architecture for Differentiated Services. RFC 2475. http://www.ietf.org
[9]	Differentiated Services Quality of Service Policy Information Base. IETF Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-diffserv-pib-09.txt

Part 2: Overview

2 Overview

The deployment of the CP-PDK programming APIs for the IXP family of network processors is shown in Figure 1. The applications invoke the NPF APIs on the control plane. The API invocation is handled by the corresponding CP-PDK module and is directed to the forwarding plane through the FP plug-in API invocation, which abstracts out the interconnection layer.

The interconnection layer can use ForCES or COPS, or shared memory in the case of co-located control and forwarding planes. For more information, refer to CP-PDK co-located transport plug-in design reference [6]. For this release, the CP-PDK uses separate optimized transport channels for control/data. The overall architecture of the forwarding plane module in the CP-PDK is shown in Figure 1.

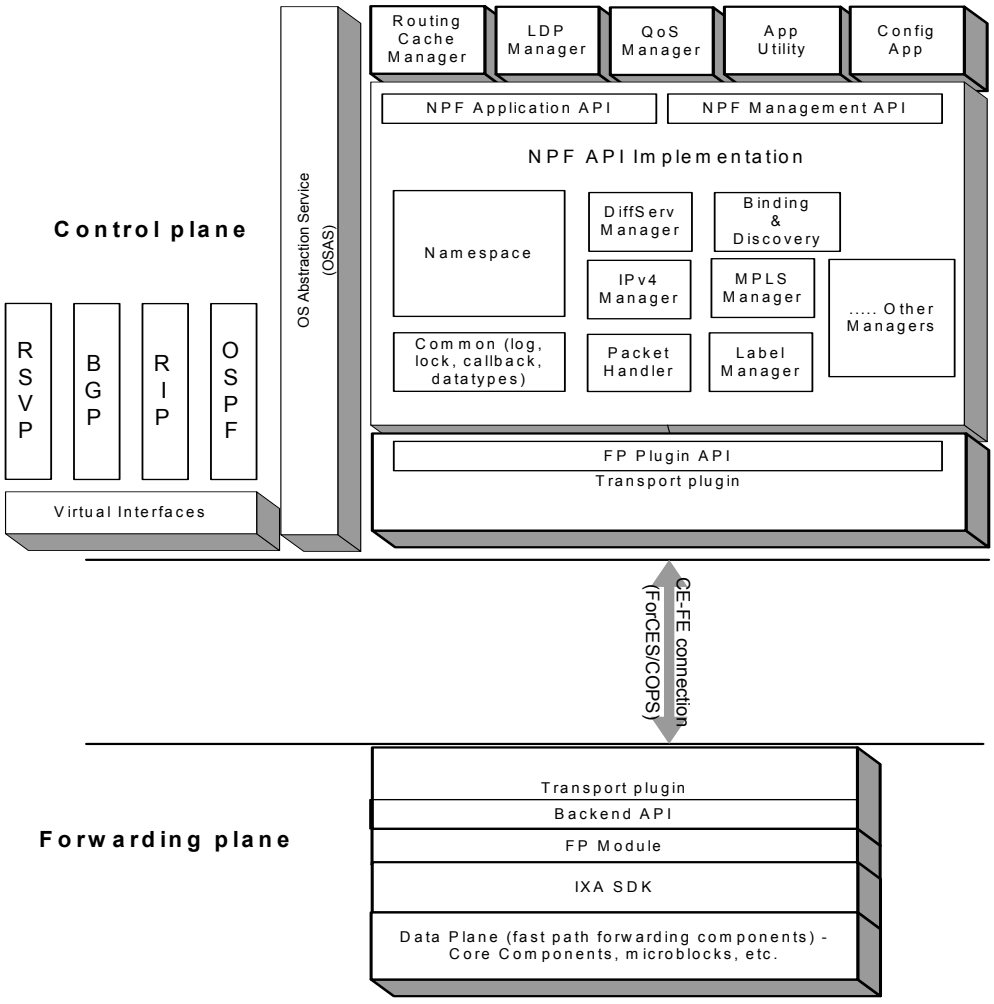


Figure 1: FP module in CP-PDK

The current design of the forwarding plane module is targeted for the Intel® IXP2XXX series of network processors making invocations into the IXA SDK 3.51 API.

2.1 Requirements

Following are the high-level requirements of the FP module:

- Must map the NPF API invocations to forwarding plane operations. The current NPF APIs includes IPv4, IPv6, and configuration and management (C&M).
- Must be completely portable to other operating systems so that it is easy to use the same design/code to a different platform on a different operating systems, such as, VxWorks* and Linux*.
- Must be modular so that any additional functionality can be added easily. For example, it should be easy to add packet handler functionality to correspond to a top level NPF packet handler API on the control plane.
- Development environment must be easy to use for external customers of the PDK. It should be easy to conditionally compile functionality and add new modules. For example, it should be easy to compile the FP module without the MPLS support, if there were no forwarding plane support for MPLS, but with QoS.
- Release 2.11 is for VxWorks/Linux and using IXA SDK 3.51.
- The translator component of each FP plug-in manager should be portable across the network processors.

2.2 Design Considerations/Assumptions

Following are some assumptions made to simplify the design.

- The FP module assumes the existence of a programming API, IXA SDK 3.5 API in the case of the Intel® IXP2400/2800, for the FP. This API presents a programming interface to the FP that is based on an NPU.
- If the IXA SDK changes, the platform-specific components of the FP module must be re-written.

2.3 Dependencies

The FP module registers call backs with he backend APIs. All messages from the FP module to the transfer plug-in and vice versa go through the backend APIs.

2.3.1 Backend API

The backend API sits between the FP module and the transport plug-in. The backend API provides the support for the following:

- Support for binding/shutdown
- NPF application level APIs – IPv4 and C&M.

- Event notification, such as, link status down/up
- Packet handling – for transporting PDUs to/from the control plane.

The backend API provides an interface that the FP invokes. This API supports the C&M of the FP, such as, adding/deleting routes and setting IP addresses. There is a definite correspondence between the NPF API, the FP plug-in API, the backend API and the FP specific APIs, such as, the CC APIs in IXA SDK 3.51. This is shown in Figure 2.

The backend API provides a query interface for the B&D functionality. Through this interface, the FE capabilities are reported to the CP. For example, the presence of MPLS data plane components makes the FP MPLS capable. This abstraction is provided to the transport plug-in through this query interface. For more details on the backend API, please refer Section 3 of this document. The transport plug-in exposes the backend API.

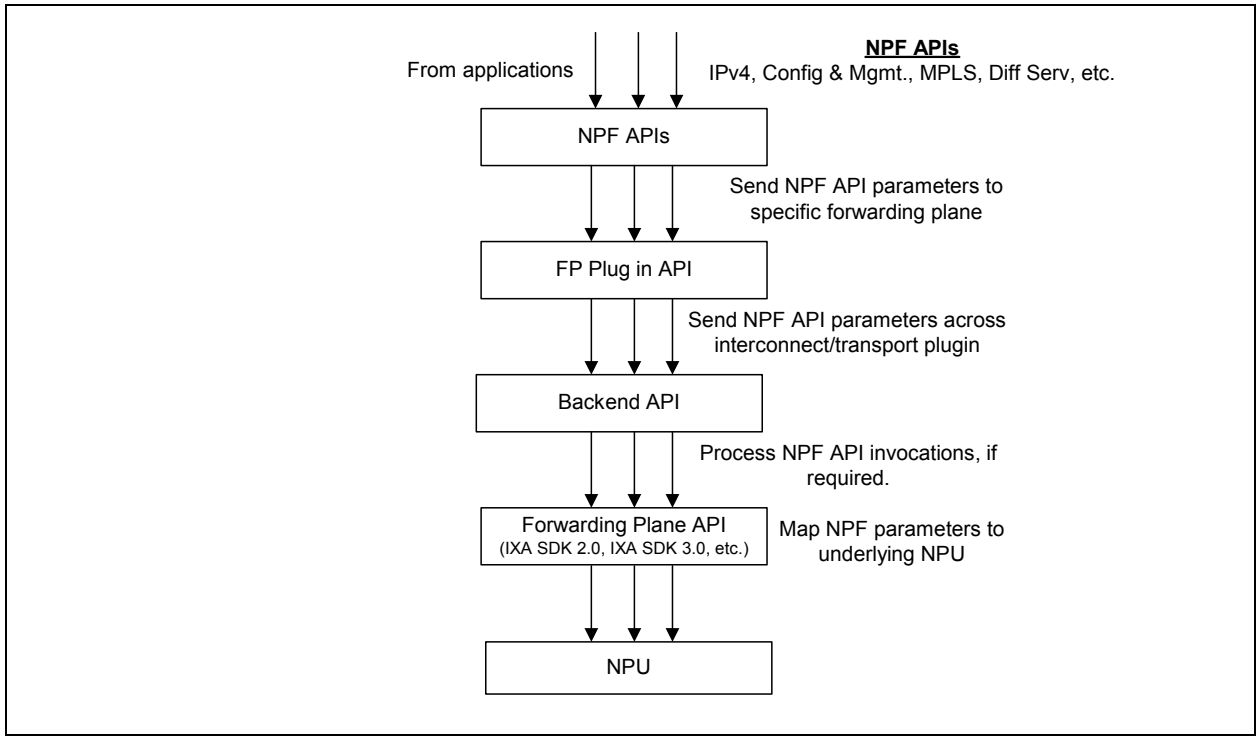


Figure 2: Translating NPF APIs to platform-specific APIs

2.4 IXA SDK

It is assumed that the reader is familiar with the Internet eXchange Architecture (IXA) and related documents.

Part 3: FP Module Design

3 FP Module Design

The FP module resides on the forwarding plane and receives the NPF API invocations from the CP through the backend API. It then maps them to the underlying forwarding components. It is specific to the forwarding plane being used.

For example, the IXP 2400-based FP exposes the IXA SDK 3.51 as its forwarding plane API (FPAPI). In the case of the IXP 2400, the FP module maps the NPF API invocation to IXP API invocations. For example, the `npf_if_set_ipaddr()` invocation of the NPF API would result in the invocation of the `ix_cc_async_set_properties()` interface by this module. In the case of the Linux*-based forwarding plane, the FP module would make IOCTL calls to the IXA SDK 3.51

It supports the binding and discovery of the forwarding plane. It provides a query interface through which the backend API can get information about the FP capabilities and the FP attributes. FE capabilities like number of ports, individual port properties such as L2 attributes (MTU), link speed, and port type are obtained during discovery.

In the case of IXA SDK 3.51, maintaining the blade ID information in the tables enables the inter-FE communication. Inter-FE can be label based and it treats the setting up of inter-FP forwarding as setting up of generic MPLS Labels on the MPLS CC.

The CP-PDK 1.05 release provided an MPLS-enabled or label-switching based solution for forwarding packets between the various forwarding planes, so as to provide the control plane applications with the semantics uniform with that of a single-box router. This module enables the setting up of inter-FP forwarding by adding the corresponding MPLS-labels to the label switching tables of the FP through the interface provided by the MPLS CC. This label-based support is not present in the CP-PDK 2.11 due to unavailability of required data plane components.

The FP module comprises of portable translators and platform-specific components. It is designed for modularity and extensibility.

3.1 High-level Overview

Figure 3: Forwarding plane module components Iiterfacing with IXA SDK 3.51 shows the internals of the FP module. The FP module has the following components,

1. FP boot manager
2. Number of FP plug-in managers, each one of which consists of the following:

- A Translator – This is a platform/OS independent component that translates NPF APIs, performs any required processing and invokes the platform-specific implementation module
- A platform-specific low-level API implementation – This module is specific to the platform being used (IXA SDK 3.51).

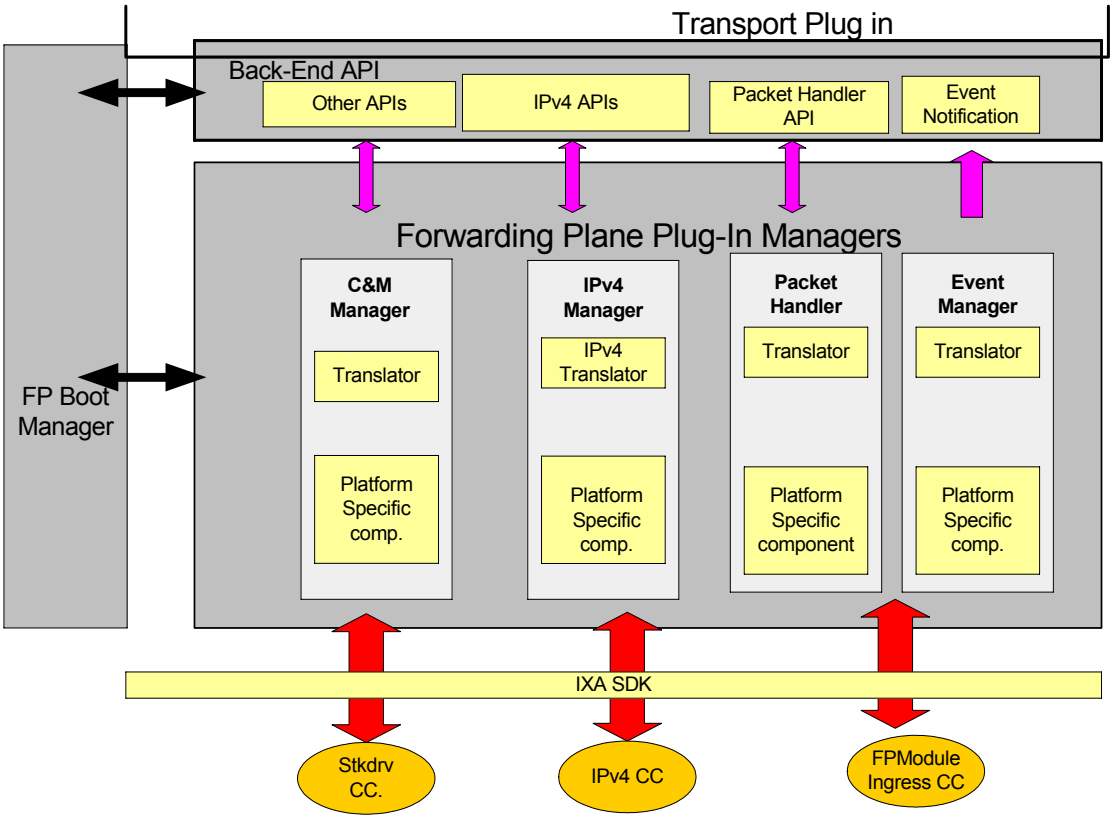


Figure 3: Forwarding plane module components literfacing with IXA SDK 3.51

The transport plug-in and the each of the FP plug-in manager translator communicate through the backend API. Each translator dispatches the NPF API invocations to the respective platform-specific component API. The FP module has the algorithms to translate NPF API functions into forwarding plane-specific ones.

For example, the add label NPF MPLS API invocation assumes that the forwarding plane has the capability to do classification and marking. This functionality is embedded into two CCs in the Forwarding plane. The FP module gets the NPF API invocation and makes the two IXA SDK API calls corresponding to the single NPF MPLS API invocation.

3.2 FP Module and IXA SDK 3.51

CP-PDK 2.11 is targeted for IXA SDK 3.51, designed to work on Intel® IXP 2400 and Intel® IXP2800 network processors. Refer to the IXA SDK documentation for more details regarding the differences between IXA SDK 2.0 and IXA SDK 3.5. One of the main components of IXA SDK 3.51 is what is termed a core component. In brief, a core component can be a packet processing entity. The PDK interfaces with several IXA SDK core components – IPv4 core component (CC), stack driver, and interface CC.

A brief introduction about the working of the core components is required here to understand some of the design decisions that have been made in the PDK, especially the FP module, to work with the IXA SDK.

3.11.1 Execution Engines and Core Components

All core components execute in the context of an execution engine (EE). A typical configuration would have an execution engine on the ingress IXP running all the core components, such as, IPv4, stack driver, and ingress RX. An execution engine is a thread of execution – it is a kernel thread in Linux* and a task in VxWorks*.

Each core component is required to export a set of user-defined functions –initialization and termination functions, one or more packet, message, or event handlers. The core component Infrastructure (CCI) provides support for handling the packets and messages in the core components. It also provides an execution engine that can be fine-tuned depending on the system requirements.

3.2.1 FP Module Execution Engine and Core Components

Most of the IXA SDK core components run on the ingress side – this is a configuration issue and does not affect the design. Components such as IPv4 and stack driver core components run on the ingress. The FP module also runs on the ingress side on the XScale. The interface transmit core component of IXA SDK and the L2 table manager run on the egress side.

In order to interact with these components, the FP module has an egress core component and an ingress mirror CC. The ingress and egress communicate with each other via the interconnect, a CSIX switch fabric. These components are shown in Figure 4. In the figure, there are two execution engines. This is a configuration issue and does not affect the design. All the core components execute in the context of one of these execution engines. The FP module ingress CC can be configured to run in either of the execution engines.

Most of the APIs exposed by the core components are asynchronous in nature. Due to this, the FP module has to register callbacks that can report any error/status conditions, or responses for queries such as statistics, interface attributes, and so on, from the underlying core component. The IXA SDK core component processes these messages and returns the responses as messages. The reply message is converted to a callback in the utility execution engine. The callback function registered by the client gets invoked.

As shown in the figure, the FP module egress CC executes in the context of the IXA SDK execution engine – the same execution engine that runs the interface transmit CC and ARP module. Here the FP module egress CC interacts with the L2 table manager also.

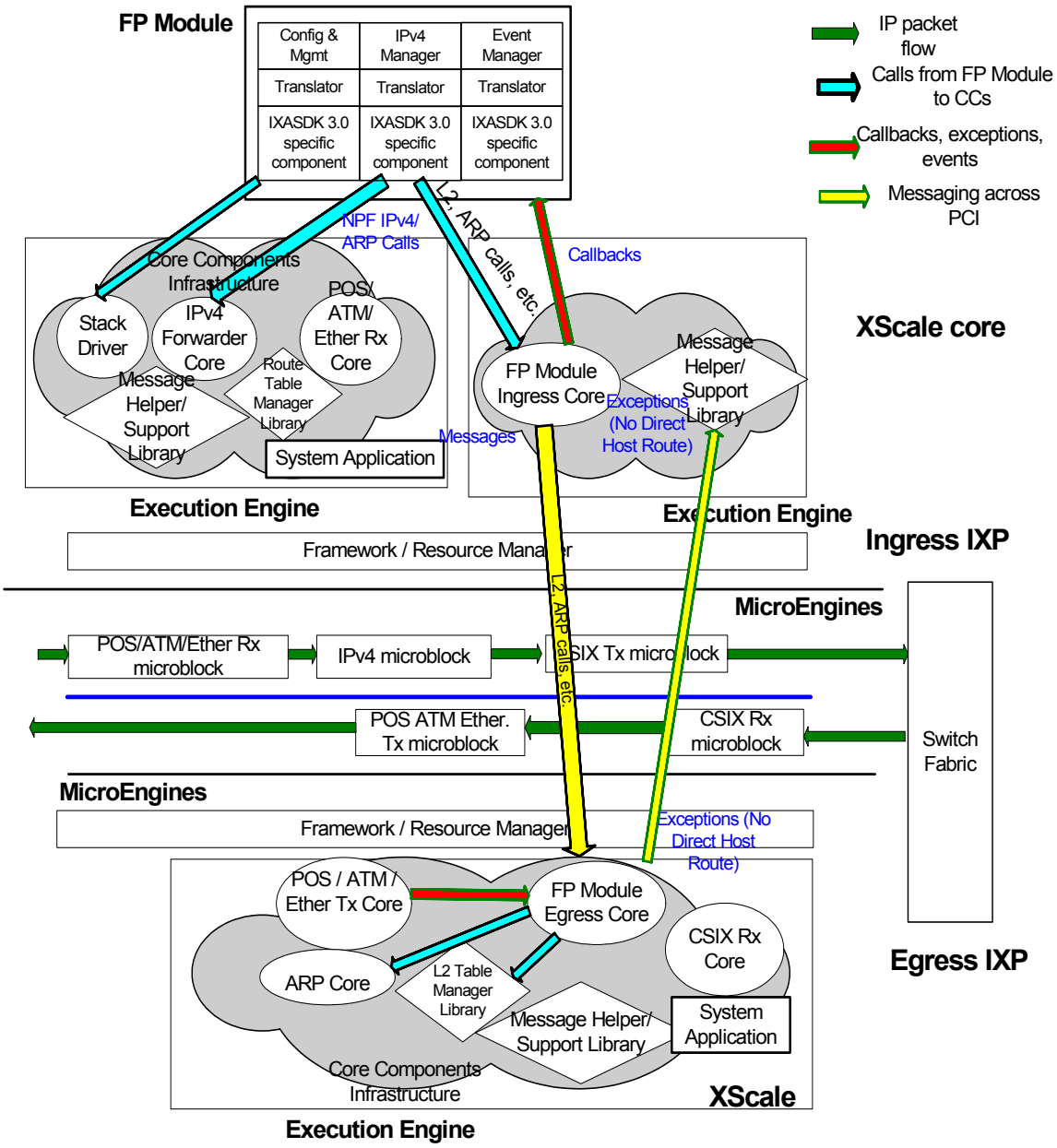


Figure 4: Overview of FP module components interaction with core components

3.3 FP Boot Manager

This section provides information on the FP boot manager The boot manager is responsible for the boot-up and the shutdown of the forwarding plane.

3.3.1 Functionality

The main responsibility of the FP boot manager is the initialization and shutdown of the forwarding plane module of the CP-PDK. It performs any platform-specific initializations and starts up all the FP plug in managers. Each manager has to expose its initialization and shutdown routines to the FP boot manager, which is described in detail in Section 3.4.4.

After initializing all the FP plug in managers, the FP boot manager runs the forwarding plane. This would result in initializing the binding process to a control plane. Once the communication with the control plane is established, the FP boot manager becomes passive and waits for a shutdown message from the control plane.

Some platform-specific initializations are performed in the FP boot manager. For PDK 1.1/2.0, an IXA SDK 3.51-specific FP boot manager performs the initializations, such as, creating an execution engine.

3.3.2 Execution Context

The FP boot manager is the main execution context for the forwarding plane module. In the case of a remote control plane, the FP boot manager has the main function.

3.3.3 Initialization

The order in which the FP plug-in managers are started could be platform-dependent. The current initialization sequence is shown in Figure 5.

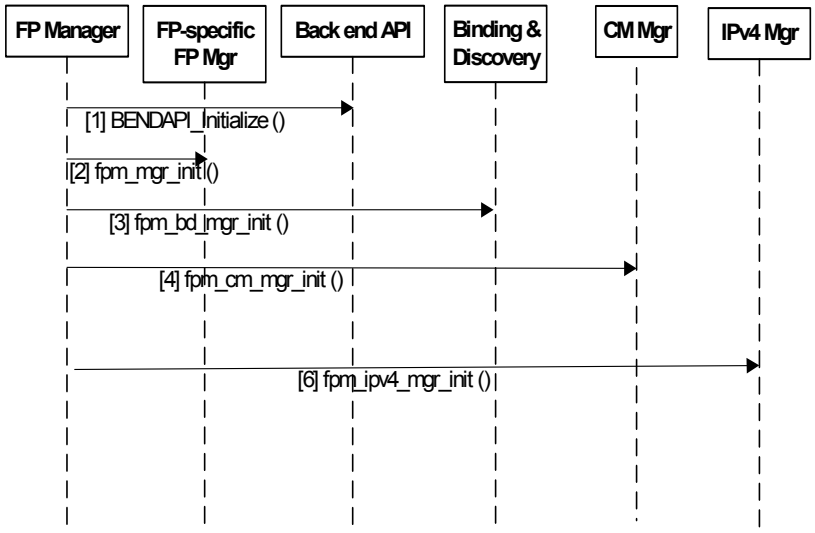


Figure 5: Forwarding plane modules initialization sequence

3.3.4 Shutdown

The order, in which the FP plug-in managers are shutdown, could be platform-dependent. The current shutdown sequence is shown in Figure 6.

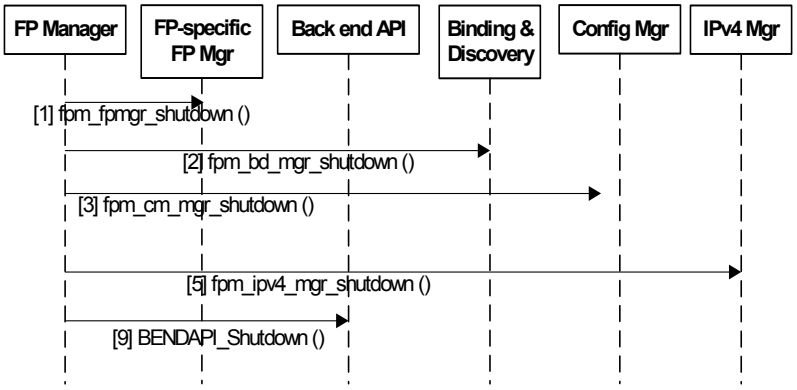


Figure 6: Shutdown sequence of forwarding plane modules

3.4 FP Plug-in Managers

Each FP plug-in manager handles a specific functional operation governed by a set of CCs. For example, the IPv4 manager handles all the IPv4 support by working with the IPv4 CC. They register the callback functions with the backend API, and handle the dispatching of the backend API invocations to the forwarding plane specific module.

As explained in Section 3.1, each manager is composed of a translator component and a platform-specific component. The translator provides a generic interface for each of the FP plug-in managers. It serves as an abstraction for the forwarding plane API specific component. This makes it easier to port all the FP plug-in managers to a different forwarding plane. For example, a Linux* based FE will have Linux-specific components talking to the translators.

The callbacks registered by each translator to the backend module are described in the external API sections, which are explained below for each of the FP plug-in managers. These external APIs are defined as down-calls. The translator also exposes up-calls, as part of the external API, which is invoked by the platform-specific component on completion of the down-call request.

On each down-call, the translator prepares a translator context comprised of a correlator from the backend API to identify the instance of the user call and a pointer to memory to be filled by the platform-specific component. This translator context is passed to the respective platform-specific component function that invokes the IXA SDK 3.51.

As a part of the IXA SDK 3.51 invocation, the platform-specific component passes a user context and a callback function wherever necessary. The user context comprises of pointers to the translator context and the request completion indicator that is used to determine a down-call request completion. For each IXA SDK 3.51 invocation a user context is allocated. The relationship between the translator and the user context is represented in Figure 7.

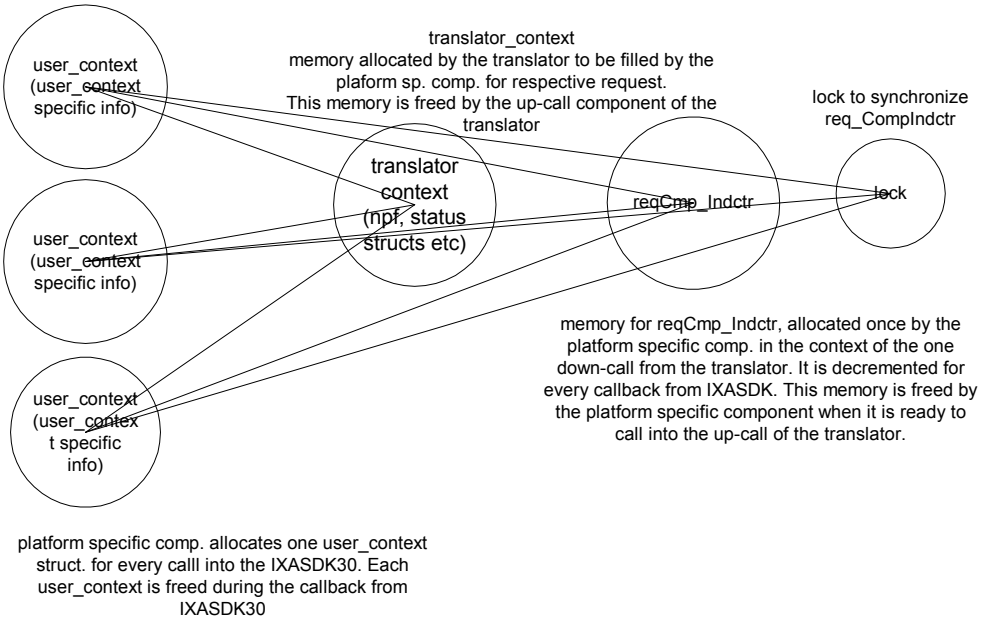


Figure 7: Relationship between translator and platform-specific user contexts

3.4.1 Requirements of Each Manager

- **Initialization and shutdown routines** – These routines are exposed to the FP boot manager and are invoked in an appropriate order during initialization and shutdown of the FP module.
- **Translator** – The translator interprets NPF messages and is independent of the platform and operating system. It should manage the memory (malloc and free) for requests that need information to be filled by the platform-specific component before the up-call.
- **Platform-specific mapping** – This converts the API parameters into the platform-specific implementation, also it hides any of the NPU-specific implementation details from its corresponding translator.

3.4.2 Translator

Each translator receives callbacks from the backend API, interprets the NPF API invocation and dispatches it to the correct platform-specific implementation. On initialization, each translator registers callbacks with the backend API. Once this is successful, it calls into the initialization routine of the respective platform-specific component. On shutdown, each translator de-registers the callback with the backend API and then calls into the shutdown routine of the respective platform-specific component. Some of the translators have extra logic, such as, the translator in MPLS manager interacts with the IPv4 manager for setting up classifier during an LSP setup.

Each of the down-calls, explained in the external API section for specific FP plug-in managers, requires the invocation of the up-call counterpart to indicate the translator request completion,

such as, `fpm_cm_mgr_set_L2_attributes()`, and `fpm_ipv4_mgr_add_prefix()`. The up-calls indicate the completion of a request to the control plane via the backend API via the `BENDAPI_Report_Status()`.

The translator performs the following steps:

1. Registers callbacks for CM down-calls with `BENDAPI`.
2. On each down-call, it parses the backend API /NPF structures. It passes the necessary information to the platform-specific component along with the translator context. If the down-call expects informationon the up-call, it allocates memory for this as part of the translator context, which is passed down to the platform-specific component. The translator context is a container for the correlator passed by the down-call from the backend API and memory for the request-specific response information for this down-call.
3. Each execution of the up-call invokes `BENDAPI_Report_Status()` to report status and to pass on any information got in the up-call. It is also responsible to free any memory allocated during the invocation of the respective down-call.

3.4.3 Platform-Specific Component

Each low-level implementation is responsible for invoking the appropriate IXA SDK 3.51 API. They basically map a NPF API invocation to the corresponding IXA SDK 3.51 API invocation. The platform-specific components expose internal APIs that are invoked by the translator down-call.

Most of the platform-specific APIs have a callback counterpart function whose address is passed while invoking the corresponding IXA SDK 3.51 API. When each of these callback functions is invoked on completion of an IXA SDK 3.51 API, it updates the translator context appropriately by filling in the request-specific info structure passed in the down-call as part of the translator context and updates the request completion counter.

Once all the callbacks of a down-call request is received, it calls into the respective translator up-call indicating completion of the request determined by the respective translator context. In certain cases, where the call into the IXA SDK 3.51 API does not take a callback function, the platform-specific component function calls the translator up-call in the same context of the down-call.

The platform-specific component performs the following functions:

1. Responsible for populating the request specific information, such as, statistics information, in the translator context for each callback from the IXA SDK.
2. Invokes the IXA SDK 3.51 as appropriate.
3. Invokes the translator up-call once all the callbacks for a down-call request are received. If the IXA SDK 3.51 API does not take a callback function, then the translator up-call is invoked on the return of the IXA SDK 3.51 API invocation.
4. Maintains the request completion indicator counter (`reqComp_Indctr`), which is updated upon every callback from the IXA SDK 3.51. The parameter is synchronized between the down-call into the IXA SDK 3.51 and the callback with the help of a lock.
5. Maintains a list of pending callbacks to help in resource management.

3.4.4 Memory Management

Each translator and platform-specific component is responsible to free the resources allocated by it. The translator is responsible to manage memory for the translator context that might include the memory for the request structure to be filled. The platform-specific component is responsible to manage the memory for the user context and other resource pointers it contains like request completion indicator.

When the FP module is being shutdown, any resources allocated for the pending requests should be freed. To achieve this each FP plug-in manger maintains a list of information on the pending callbacks. This list is updated on every callback from the IXA SDK 3.51. On the shutdown of each manager, the pending list is walked through and the resources are freed and the corresponding up-call is invoked, freeing up the resources tied to the respective translator context.

In a situation when some pending callbacks from the IXA SDK 3.51 never get invoked, the platform-specific components need to free up the resources for the respective user contexts. The translator components should free up their respective memory allocated to the translator context. Each entry in the pending callback list comprises of a timestamp, a type to specify the down-call requesting function, and the pointer to the user context.

A thread checks the status of the pending callbacks at regular intervals and frees up those expired pending callback resources and invokes the respective translator up-call with a failure result status. The translator reports the status to the backend module, and free up its own resources.

3.4.5 Exception Handling

The FP module converts exceptions to actions raised by the forwarding plane into the CP-PDK understandable exceptions. For example, in case of the IXP API, an add prefix call can fail generating an `IXAPI_FAILURE`. This has to be converted to a CP-PDK defined exception, such as, `FPPI_FAILURE`, and passed on to the backend API.

The subsequent sections describe in detail each of the FP plug-in managers currently implemented in the PDK.

3.5 FP Module on Linux*

In order to have the Forwarding Plane ported to Linux or any other standard OS, some issues need to be considered. For example, some of the things that need to be done are:

- **IPv4:** Map all IPv4 calls to interface with the core component in Linux kernel. For example, the NPF IPv4 add prefix call might translate to making a write system call in Linux to add this route to the FIB (Forwarding Information Base). The IPv4 Manager has to handle this.

- **C&M:** C&M handles configuring of ports and/or interfaces. For example, setting IP address, MTU, link speed, and so on The C&M must make all core component API calls through write system call into the Linux kernel.

- **Handling packets:** Data packets that are destined to control plane applications should be redirected through the PDK. There can be two scenarios: a) The packet may be meant for a legacy application using the socket() interface b) The packet may be meant for an application that makes

use of NPF Packet Handler APIs. PDK 2.11 supports the first case only. Similarly, PDK must capture all outgoing packets from the Linux IP stack and tunnel them to the Forwarding Plane.

The above examples of what needs to be done are not exhaustive. This just aims to show how the FP Module can be easily extended to work on any traditional OS.

3.5.1 FP Module Design for Linux

As the IXA SDK 3.51 for the Linux operating system is located in the kernel space, the FP Module needs to interact with the Linux kernel. The OS-specific component of each of the FP Plug-in managers resides in the kernel space as shown in . Each translator interacts with a *Shim Layer* to communicate to its respective OS-specific component. This shim layer exposes the cumulative APIs of all the core components, or in other words, the shim layer has proxy implementations for each of the core component APIs. When a *down-call* comes via the translator the appropriate proxy API on the shim layer is invoked. The proxy core component API does the following:

- Creates shim layer context for the IXA_SDK API call (This context maintains information such as the client regiatered call back function to be called, client context information etc.)
- Passes the context to the OS-specific component by invoking a write system call

Since the OS-specific component must expose system calls such as read and write, it is implemented as a character device.

Since the core components are in kernel space, the core components cannot directly call the client defined callback function. Therefore, the shim layer passes its own function as the callback function to the core component. We refer to these functions as *proxy callback* functions. Proxy callback functions format a response message and put it in a queue maintained by the shim layer in the kernel.

The reader thread of shim layer *polls* on a callback descriptor (socket or file descriptor) by blocking on a read(). Whenever a response message is received, the reader thread retrieves the corresponding context and invokes the client registered callback. Figure 8 shows the architecture of the FP module for the Linux OS.

The support for this is implemented in subsequent releases of the CP-PDK (post 2.0) and IXA SDK 3.51.

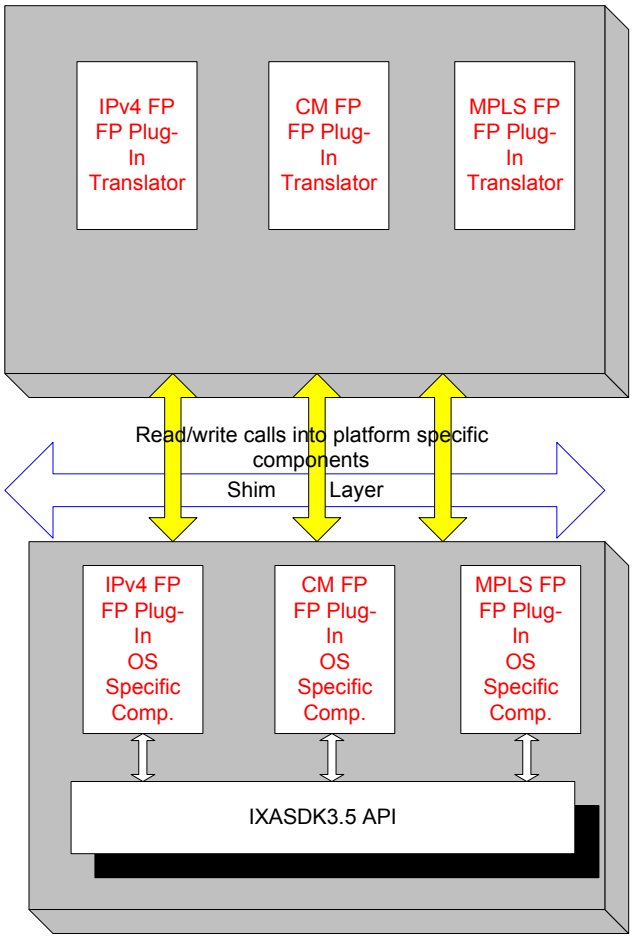


Figure 8: FP Module for Linux

3.5.2 Initialization

The initialization function for the Shim layer is executed when the kernel module is inserted. This function registers the character device with the kernel. Since the shim layer exposes APIs for all of the core components there is only one character device.

The pseudocode for the module initialization function is as follows:

```
/* The following is the file operations structure for the
character device. The device implements read() and write() calls
*/

static struct file_operations shim_fops = {
        read: sh_read,
        write: sh_write,
};
```

```
int init_module (void)
{
    Call register_chrdev( ) with major number, character device
name and file operations structure as inputs;
    Initialise the queue;
}
```

Once the character device is created, a node must be created in /dev/ directory with a command as follows (Major number is assumed to be 224 and minor number 0):

mknod /dev/shim c 224 0

Now that the node is created any user mode application can open this device, read and write into it.

3.5.3 Shutdown

The shutdown function of the shim layer is invoked when the module is removed from the kernel. This function unregisters the character device from the kernel.

The pseudocode for the cleanup function is as follows:

```
void cleanup_module (void)
{
    Call unregister_chrdev( ) function with major number and
device name as parameters;
    Release allocated memory, eg: queue etc;
}
```

3.5.4 Data Structures

For each request that is sent to the core component, the Shim layer maintains a context in the user space. This context maintains information such as the module to which this request is directed, the action that needs to be done etc. The component name and the action uniquely identify the core component function that must be invoked. This context is used when the callback function is invoked. However, when a callback function is invoked corresponding to an event there will not be any context to it. Following is the structure that is maintained for each request.

```
typedef struct
{
   SH_COMPONENT           compName;      /* Indicates the core
component to
                                             which
                                             this request is
directed */
    void*                 userContext;   /* Client information */
    void*                 callback;      /*  Client defined
callback
```

```
                                        function */
    union                              /* The action identifies
the core
                                           component call
                                           to be made */
    {
         SH_IPV4_ACTION          ipv4Action;
         SH_FPM_ACTION           fpmAction;
         SH_STKDRV_ACTION        stkdrvAction;
     } u;
     ix_error            requestStatus;  /* The error code will be
set if
                                           an error is
encountered
                                           in the request path */
     ix_error            responseStatus; /* The error code will be
set if
                                           the callback
                                           returns an error */
     void*               params[MAX_NUM_PARAMS];  /* Pointers to
                                                     parameter
structures */
} SH_Context_t;
```

The component name and the action parameters uniquely identify the core component function that will be invoked. The parameters to the core component call are passed as pointers in the params field of the above structure.

The following is the data structure used to hold the callback data in the kernel queue. It is explained in detail later.

```
typedef struct
{
    SH_ResponseMessageHeader respHeader;  /*Header information */
    struct
    {
         ix_uint32         length;       /* Number of bytes in the
next
                                            field */
         void*             responseData; /* Call back data */
     }   outData[MAX_NUM_RESP];
} SH_GeneralResponseMessage;

typedef struct
{
   SH_COMPONENT           compName;       /* Indicates the core
component
```

```
                                                  that originated
                                                  this response */
    void*                 shContext;      /* Client information */
    ix_error              responseStatus; /* This error code will
be set
                                                   if the callback
returns error */
    union                                 /* The action identifies
the
                                                  core component call
                                                  made */
    {
         SH_IPV4_ACTION           ipv4Action;
         SH_FPM_ACTION            fpmAction;
         SH_STKDRV_ACTION         stkdrvAction;
     } u;
} SH_ResponseMessageHeader;
```

The following enumeration is used to list down the core components supported.

```
typedef enum {
    SH_IPV4_CC,
    SH_FPM_CC,
    SH_FPM_DATA_CC,
    SH_STKDRV_CC
} SH_COMPONENT;
```

Data Structure for IPv4 Manager

The IPv4 Manager can initiate one of the following actions.

```
typedef enum {
    SH_IPV4_ASYNC_ADD_ROUTE,
    SH_IPV4_ASYNC_DEL_ROUTE,
          SH_IPV4_ASYNC_UPDATE_ROUTE,
    SH_IPV4_GET_ROUTE,
         SH_IPV4_ASYNC_LOOKUP_ROUTE,
    SH_IPV4_ASYNC_FLUSH_PREFIX,
    SH_IPV4_ASYNC_FLUSH_NHOP,
    SH_IPV4_ASYNC_UPDATE_NHOP,
    SH_IPV4_ASYNC_ADD_NHOP,
    SH_IPV4_ASYNC_DEL_NHOP,
    SH_IPV4_ASYNC_GET_NHOP
} SH_IPV4_ACTION;
```

Following are the actions that can be initiated on FPM core component.

```
typedef enum {
     SH_FPM_REGISTER_LINKST,
     SH_FPM_REGISTER_IPV4EXCP,
           SH_FPM_REGISTER_PKTHDLR,
     SH_FPM_DEREGISTER_LINKST,
     SH_FPM_DEREGISTER_IPV4EXCP,
           SH_FPM_DEREGISTER_PKTHDLR,
     SH_FPM_ADD_L2ENTRY,
           SH_FPM_ASYNC_ADD_L2ENTRY,
     SH_FPM_DEL_L2ENTRY,
           SH_FPM_ASYNC_DEL_L2ENTRY,
     SH_FPM_ADD_ARPENTRY,
           SH_FPM_ASYNC_ADD_ARPENTRY,
    SH_FPM_DEL_ARPENTRY,
         SH_FPM_ASYNC_DEL_ARPENTRY,
        SH_FPM_FLUSH_ARP,
    SH_FPM_EVENT_LINKST,
         SH_FPM_EVENT_IPV4EXCP,
       SH_FPM_CONFIG_CPADDR
} SH_FPM_ACTION;
```

Data Structure used by Configuration Manager

The Configuration Manager may initiate one of the following actions on the stack driver core component.

```
typedef enum {
   SH_STKDRV_GET_NUMPORTS,
   SH_STKDRV_GET_PROPERTY,
   SH_STKDRV_SET_PROPERTY
} SH_STKDRV_ACTION;
```

3.5.5 Invocation of a SDK call

The following are the steps involved while making an asynchronous SDK call.

- The backend API makes a call into the FP Module translator (eg: fpm_ipv4_mgr_add_prefix)
- The FP Module translator calls the platform specific function (eg: ipv4_add_prefix)
- The platform specific function calls the core component function - this function is proxy to the actual IXA SDK API and is implemented in the user space.
- The proxy core component function constructs a shim layer context and makes a write system call to invoke the core component API

- The shim layer calls SDK function (eg: ix_cc_ipv4_async_add_route) passing its own (eg: sh_ipv4_cb) callback function
- The callback function in the shim layer (eg: sh_ipv4_cb) gets called which puts the callback data into the kernel queue and makes the character device readable
- The user thread that blocks on the read() call is unblocked
- The user thread reads a SH_GeneralResponseMessage structure
- The user thread invokes the client registered callback function

In case of a synchronous SDK call, the core component function is called in the same thread of execution as the write system call and there is no response message in this case.

3.5.6 External APIs of the Character Device

The FP Module translator calls the write() function on the character device to invoke a SDK call. The pseudocode for the write function of the shim layer is as follows:

```
uint32_t sh_write( )
{
    reqstMessage = (SH_Context_t *)data;
             /* Determine the core component on which the function
is
                to be invoked */
    switch (reqstMessage->compName)
    {
        case SH_IPV4_CC:
            switch (reqstMessage->u.ipv4Action)
            {
                case SH_IPV4_ASYNC_ADD_ROUTE:
                          Call function
ix_cc_ipv4_async_add_route() with
                          sh_ipv4_cb as the callback function and
parameters
                          retrieved from reqstMessage->params[0],
                          reqstMessage->params[1] etc.

                          If the core component function call
fails,
                          set ix_error in reqstMessage-
>requestStatus and
                          return -1 to indicate error;
                          break;
                case SH_IPV4_GET_ROUTE:
```

```
                        Call function ix_cc_rtmv4_get_route()
with
                        sh_ipv4_cb as the callback function and
parameters
                        retrieved from reqstMessage->params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call
fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                case SH_IPV4_ASYNC_DEL_ROUTE:
                        Call function
ix_cc_ipv4_async_delete_route() with
                        sh_ipv4_cb as the callback function and
parameters
                        retrieved from reqstMessage->params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                case SH_IPV4_ASYNC_FLUSH_PREFIX:
                        Call function
ix_cc_ipv4_async_purge_routes() with
                        sh_ipv4_cb as the callback function and
parameters
                        retrieved from reqstMessage->params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call
fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                case SH_IPV4_ASYNC_UPDATE_NHOP:
                        Call function
ix_cc_ipv4_async_update_next_hop()
                        with sh_ipv4_cb as the callback  function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call fails,
```

```
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                case SH_IPV4_ASYNC_ADD_NHOP:
                        Call function
ix_cc_ipv4_async_add_next_hop()
                        with sh_ipv4_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                case SH_IPV4_ASYNC_GET_NHOP:
                        Call function ix_cc_rtmv4_get_next_hop()
                        with sh_ipv4_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                case SH_IPV4_ASYNC_DEL_NHOP:
                        Call function
ix_cc_ipv4_async_delete_next_hop()
                        with sh_ipv4_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                         reqstMessage->params[1] etc.

                         If the core component function call
fails,
                         set ix_error in reqstMessage-
>requestStatus and
                         return -1 to indicate error;
                         break;
                case SH_IPV4_ASYNC_FLUSH_NHOP:
                        Call function ix_cc_ipv4_async_purge_rtm()
                        with sh_ipv4_cb as the callback function
and
```

```
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error.
                        break;

                default:
                       /* Error: Not a valid action */
              }
             break;   /* IPv4 case ends here */

        case SH_FPM_CC:
                 /* Call the core component function based on
action
                    field of fpmStruct */
             switch (reqstMessage->u.fpmAction)
            {
                 case SH_FPM_REGISTER_LINKST:
                                                        Call
function ix_cc_fpm_register_link_status_cb() with

sh_onCB_cc_link_status as the callback function;

                                                        If the core
component function call fails, set
                         ix_error in reqstMessage->requestStatus
and
                         return -1 to indicate error;
                         break;
                                    case
SH_FPM_REGISTER_IPV4EXCP:
                                                        Call
function ix_cc_fpm_register_ipv4_exception_cb() with

sh_onCB_cc_ipv4_exception  as the callback function;

                                                        If the core
component function call fails, set
                         ix_error in reqstMessage->requestStatus
and
                         return -1 to indicate error;
                         break;
                                        case SH_FPM_REGISTER_PKTHDLR:
                                                      Call function
ix_cc_fpm_register_pkt_hdlr_cb() with
```

```
sh_write_packet  as the callback function;

                                                    If the core
component function call fails, set
                        ix_error in reqstMessage->requestStatus
and
                        return -1 to indicate error;
                        break;
                                    case
SH_FPM_DEREGISTER_LINKST:
                                                    Call
function ix_cc_fpm_deregister_link_status_cb().

                                                    If the core
component function call fails, set
                        ix_error in reqstMessage->requestStatus
and
                        return -1 to indicate error;
                        break;
                                    case
SH_FPM_DEREGISTER_IPV4EXCP:
                                                     Call
function ix_cc_fpm_deregister_ipv4_exception_cb();

                                                     If the core
component function call fails, set
                        ix_error in reqstMessage->requestStatus
and
                        return -1 to indicate error;
                        break;
                                    case
SH_FPM_DEREGISTER_PKTHDLR:
                                                    Call
function ix_cc_fpm_deregister_pkt_hdlr_cb().

                                                     If the core
component function call fails, set
                                                     ix_error in
reqstMessage->requestStatus and
                                                     return -1
to indicate error;
                                                     break;
                case SH_FPM_ASYNC_ADD_L2ENTRY:
                        Call function
ix_cc_fpm_async_add_l2_entry()
                        with sh_fpm_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.
```

```
                        If the core component function call
fails, set
                        ix_error in reqstMessage->requestStatus
and
                        return -1 to indicate error;
                        break;
                case SH_FPM_ASYNC_DEL_L2ENTRY:
                        Call function
ix_cc_fpm_async_del_l2_entry()
                        with sh_fpm_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call
fails,
                        set ix_error in reqstMessage-
>requestStatus and
                        return -1 to indicate error;
                        break;
                 case SH_FPM_ASYNC_ADD_ARPENTRY:
                        Call function
ix_cc_fpm_async_add_arp_entry()
                        with sh_fpm_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call
fails, set
                        ix_error in reqstMessage->requestStatus
and
                        return -1 to indicate error;
                        break;
                case SH_FPM_ASYNC_DEL_ARPENTRY:
                        Call function
ix_cc_fpm_async_del_arp_entry()
                        with sh_fpm_cb as the callback function
and
                        parameters retrieved from reqstMessage-
>params[0],
                        reqstMessage->params[1] etc.

                        If the core component function call
fails, set
                        ix_error in reqstMessage->requestStatus
and
                        return -1 to indicate error;
```

```
                              break;
                                                  case SH_FPM_EVENT_LINKST:
                              Call function
ix_cc_fpm_register_link_status_cb
                              with sh_cc_link_status as the callback
function.

                              If the core component function call
fails, set
                              ix_error in reqstMessage->requestStatus
and
                              return -1 to indicate error.
                              break;
                        default:
                              /* This action is not supported by the
FPM CC */
                  }
                  break;
            /* Handle other core component functionality here */
            default:
                    /* There is no support for this core component */
      }
             Return the number of bytes written – this is same as the
number
             of bytes passed in the input buffer;
}
```

Each OS specific module registers its own callback function with the core component that proxies the client registered callback function. Following functionality must be executed by each such callback function:

- Allocate kernel memory for a structure of type SH_GeneralResponseMessage
- Fill in the above structure including the response data
- Put the message in the kernel queue
- Make the character device readable by releasing the semaphore

Note: The above method of passing the response data to the user space requires two copies – the first one is done inside the proxy callback function that copies the callback response into the kernel queue and the second copy is done from the kernel queue to the user space when the read system call is called. A different approach can be that the user space module passes a pointer to a pre-allocated memory along with the request itself and the proxy callback function copies the callback response directly into the user space. The second approach requires only one copy. However, the second approach cannot handle events. Since the first approach can handle both asynchronous calls and events, it has been adopted in the design.

sh_ipv4_cb() is the callback function in the kernel for all IPv4 CC calls. The pseudocode for this function is given below:

```
ix_error sh_ipv4_cb( )
{
     Set the error code in responseStatus in shim context;
     Fill in the fields of a structure of type
SH_GeneralResponseMessage;
     Put the structure in the queue;
     Make the character device readable by executing a give
operation on the semaphore;
}
```

```
sh_fpm_cb() is the callback function in the kernel for all FPM CC
calls.
ix_error sh_fpm_cb( )
{
     Set the error code in responseStatus in shim context;
     Fill in the fields of a structure of type
SH_GeneralResponseMessage;
     Put the structure in the queue;
     Make the character device readable by executing a give
operation on the semaphore;
}
```

The following is the sh_read() function implemented in the character device.

```
uint32_t sh_read( )
{
     Check if there is any response queued, if present
               take first available response,
               copy it to the user buffer and
               dequeue the node;
    If there is no response queued,
               block on the semaphore by executing a take
operation;
               When unblocked,
                    take the available response and pass it to the
user;
}
```

3.5.7 Reader Thread

There must be a thread created in the user space that blocks on read( ) on the character device. This thread is created in the initialization function of the shim module.

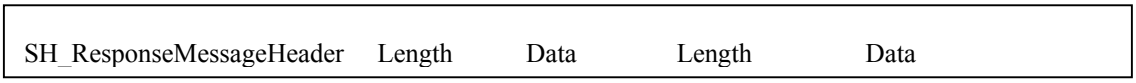
```
NPF_RET sh_init( )
```

```
{
    Open the character device and get the file descriptor;
    Create a thread and pass the file descriptor on to the thread;
}
```

The user thread that blocks on the read() call gets unblocked when the proxy callback functions put a message in the queue. The thread then reads the message. There can be one or more callback functions per core component.

The thread starts its execution at the following function. This function keeps reading from the character device as far as data is available on the device. If there is no data available, the thread blocks on the read() function call.

The buffer that the read thread gets from the kernel has data in the following format:




```
uint32_t sh_reader_thread( )
{
    char         buf[MAX_RESPONSE_SIZE];   /* Static buffer
allocated to
                                              read the response
                                              data from the kernel
*/
    SH_ResponseMessageHeader*        respHdr;

    Block on read() function call;
    /* The following functionality is executed when read() is
unblocked */
    Get callback response data in a buffer;
    respHdr = (SH_ResponseMessageHeader *)buf;
    switch(respHdr->compName)
     {
          case SH_IPV4_CC:
           If the respHdr->u.ipv4Action denotes an event
                Get the callback function from the list of
functions
                based on component name
                and action and call it;
           else
               Call the function denoted by respHdr-
>shContext.callback;
          break;
          case SH_FPM_CC:
            If the respHdr->u.fpmAction denotes an event
                Get the callback function from the list of
functions
```

```
                    based on component name
                    and action and call it;
               else
                   Call the function denoted by respHdr-
>shContext.callback;
               break;
               default:
                   /* This core component is not supported */
         }
       Free the shim context;
       Go back and block on read again;
}
```

3.5.8 Handling Events

The user space shim layer maintains a list of callback functions for different events. Each node in the list will have the following information:

```
typedef struct
{
   SH_COMPONENT           compName;      /* Indicates the core
component
                                            that originated
                                            this response */
   void*                  callback;      /* Callback function to
be called */
    union                                 /* The action identifies
the
                                            core component call
                                              made */
    {
         SH_IPV4_ACTION          ipv4Action;
         SH_FPM_ACTION           fpmAction;
         SH_STKDRV_ACTION        stkdrvAction;
     } u;
} SH_EventInfo_t;
```

When events such as link up/down, IPv4 direct host attach are generated, they will be delivered to the user similar to the way a callback response is delivered. However, in the case of events there is no shim context. Based on the component name and the action, the user defined callback function will be chosen and called.

3.5.9 Handling Data Packets

The data packets are handled in the linux kernel itself. A network device is implemented as a kernel module that registers with the kernel for IP protocol IPPROTO_VIP. All the tunneled data packets that are exchanged between

the Control Plane and the Forwarding Plane are sent with protocol set to IPPROTO_VIP. The data packets exchanged between the control plane and the forwarding plane have the format as shown below:

IP Header 2	Meta Data	IP Header 1	IP Payload

IP Header 2 will have the protocol set to IPPROTO_VIP. The meta data contains the packet length and the port information.

The network device must support the following IOCTL commands:

SIOCCPADDRESS: Used to initialize the Control Plane IP address

SIOCREGISTERHNDLR: Used to register a callback function with the fpm

The Packet Handler module in the User Space is responsible for initializing the Control Plane IP address in the network device and to have the network device register the callback function with the fpm.

When a tunneled packet is received from the Control Plane, the receive function of the network device will be called.

The packet that is delivered to the receive function will have the following format:

Meta Data	IP header	Payload

The pseudocode for the receive function is as follows:

```
int vipip_rcv ( struct sk_buff *skb)
{
    Strip off the metadata from the packet;
    Call ix_cc_fpm_sync_send_packet to deliver the packet to the
core component;
}
```

vipip_tunnel_write() is the function that will be registered with the fpm module.

The packet that is received at the vipip_tunnel_write() function will look as follows:

IP Header	Payload

The pseudocode for the function will be as follows:

```
int vipip_tunnel_write( )
```

```
{
      Add the metadata to the packet;
      Add an IP header with the protocol set to IPPROTO_VIP;
      Send the packet by calling IPTUNNEL_XMIT() macro;
}
```

3.5.10 Memory Management

When the FP Module issues a request to the character device, the character device will not copy the parameters of the SH_Context_t again but will pass them to the core component functions. There is no need to make a copy of the request because the write( ) call returns only after the core component call returns. The FP Module that has allocated memory for the request, must free them.

When the user thread issues a read() call, it must pass a pointer to a user space buffer. The character device will copy the response data into the user memory.

3.6 Binding and Discovery

This section gives an overview of the B&D FP plug-in manager.

3.6.1 Functionality

This module is responsible for:

- Initializing the communication with a control plane, sending a bind request
- Reporting of the FP capabilities and attributes to the transport plug-in through the backend API.

The different capabilities reported by this module are:

- FE-wide capabilities - Number of ports, individual port attributes like MTU, and link speed

Since PDK 2.11 does not contain QOS/MPLS data plane, the QOS/MPLS capabilities are not reported.

3.6.2 Execution Context

This module is initialized by the FP boot manager and runs in the same context. Once all the FP module components are initialized, the FP boot manager initiates the bind sequence by calling into this module. The capability reporting is in the context of the backend API, which receives responses from the Control Plane.

3.6.3 Initialization

The FP boot manager initializes this module during the start up of the forwarding plane. Since this module registers callbacks with the backend API, it has to be initialized after the backend API (transport plug-in).

```
FPPI_RET fpm_bd_mgr_init {
```

register callback with backend API for BIND RESPONSE message

register callback with backend API for UNBIND message

register callback with backend API for CAPABILITY REPLY message

```
}
```

3.6.4 Shutdown

This module is shutdown by the FP boot manager. Since this module should de-register the callbacks with the backend API, it must be shutdown before backend API (transport plug-in).

```
FPPI_RET fpm_bd_mgr_shutdown {
```

De-register callback with backend API for BIND RESPONSE message

De-register callback with backend API for UNBIND message

De-register callback with backend API for CAPABILITY message

```
}
```

3.6.5 External API

The FP boot manager invokes the `fpm_bd_mgr_start()` function after the performance of all initializations. This function calls into the configuration and management manager (CMM) to get FE-wide capabilities. This call also returns some system-wide IPv4 properties, such as, whether the Equal Cost Multi Path (ECMP) routing is supported by the underlying data plane. As mentioned earlier, the calls into the CMM are all asynchronous. Therefore, B&D registers a callback that is invoked when the CMM gets the properties from the stack driver. When the CMM has all the FE properties from the Stack driver, it invokes the callback registered by B&D.

```
fpm_bd_mgr_start ()
{
fpm_cm_mgr_get_fecaps (fe_name);
}
```

The callback function mentioned below is called by the CMM when it has retrieved all the FE capabilities. The B&D stores the FE capabilities and sends a bind request to the CE with the FE ID, such as, blade ID from the underlying IXA SDK.

```
bd_mgr_report_fecaps_cb (uint32_t fe_id, FPPI_FECaps* fecaps)
```

{

store `fecaps` (this is needed later to send a capability report)

send bind request (fe_id);

}

3.6.6 On Bind Response

This function is registered as a callback with the backend API for receiving the bind response from the control plane. The backend API invokes this function in response to a bind request.

The B&D now sends a capability report message through the backend API reporting the FE capabilities that are stored earlier.

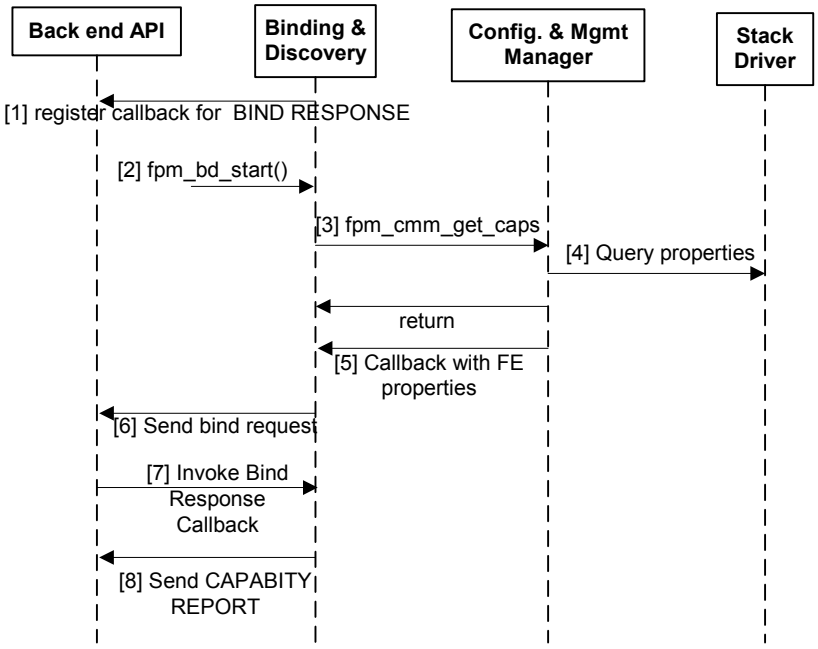


Figure 10: Querying capabilities from FP plug in managers

3.7 Packet Handler

This section gives an overview of the packet handler in the FP module.

3.7.1 Functionality

In the case of physically separated control and forwarding planes, routing protocols and/or applications might be executing on the remote control plane. To an external device, the separated CE-FE looks like a single box with a set of interfaces. The packets destined to the

applications/protocols running on the control plane reach the forwarding plane first, where the decision has to be taken as to where these packets should be sent.

For example, in the case of OSPF running on the control plane, route updates, and hello messages, the OSPF expects must be captured and tunneled to the control plane, where the OSPF sockets are open. The details of this support can be found in the protocol support services design reference, and only some relevant sections are touched upon here for clarity.

Support for packet handling on the forwarding plane consists of the following:

- **Capture locally directed/control packets:** This has to be done in a platform-specific manner. Subsequent sections will describe the support on IXA SDK 3.51 (for IXP 2400 and 2800).
- **Tunnel packets to the control plane:** This is done in a fairly simple manner. The packet handler of the FP module gets the packet and sends it to the control plane through the transport plug-in using the backend API.
- **Receive outgoing packets/PDUs from control plane:** Packets going out of the control plane arrive through the transport plug-in and reaches the packet handler. This has to be sent out now on the correct egress interface.
- **Transmit PDUs/packets out of the correct egress interface:** This is platform specific. The details of how this is done for different platforms are explained subsequently.

For a co-located VxWorks-based CP and FP, the packets going out from or coming into the PDK client stacks are handled completely by the IXA SDK 3.51 components through the local VIDD and stack driver core components.

For CP-PDK 2.11 and remote control plane stacks, locally destined packets from the stack driver reach the FP module ingress core component. From the FP module ingress core component, the packets are tunneled to the control plane. The details of this tunneling protocol can be found in the transport plug-in design reference. Outbound packets from the control plane reach the FP module ingress CC and are sent to the stack driver for transmission.

3.8 Configuration and Management Manager (CMM)

This section gives an overview of the C&M FP plug-in manager.

3.8.1 Functionality

The CMM is responsible for setting and getting properties like IP address, MTU, and link-speed. It is responsible for getting the capabilities of the FE as a part of the FE B&D process.

3.8.2 Execution Context

All the C&M manager down-calls execute in the context of the backend API callback functions registered during the initialization time. Where the calls into the IXA SDK 3.51 are synchronous, then the up-calls execute in the context of the backend API callback functions, for example, `fpm_cm_mgr_set_L3_attrib().` If the calls into the IXA SDK 3.51 are asynchronous then up-calls execute in the context of the callbacks from the IXA SDK 3.51, for example, `fpm_cm_mgr_get_fecaps().`

3.8.3 Initialization

The FP boot manager invokes the initialization function `fpm_cm_mgr_init ()`. This function registers the callbacks with the backend API for setting and getting interface properties and statistics and invokes the platform-specific initialization function. The platform-specific component queries for the maximum number of ports and stores the value into a global variable.

3.8.4 Shutdown

The FP boot manager invokes the shutdown function `fpm_cm_mgr_shutdown ().` This function de-registers the callbacks from the backend API and invokes the platform-specific shutdown function.

3.8.5 Data Structures

This section provides information on the data structures used by the C&M FP plug-in manager.

3.8.5.1 Translator-Related Structures:

Stores the generic translator's context:

```
typedef struct
{
      uint32_t cbCorrelator;  // Higher-level instance differentiator, opaque to the FP module.
} cm_TranslatorContext_t;
```

Stores the translator's context related to getting num of ports:

```
typedef struct
{
      char get_fecaps;  // Boolean, 1 indicates fecaps is to be retrieved
      char *fename;  // pointer to the name of the FE
      uint32_t *port_num; // memory allocated for holding port number
      uint32_t cbCorrelator; // Higher-level instance differentiator, opaque to the FP module.
```

```
} cm_portnum_TranslatorContext_t;
```

Stores the translator's context related to getting the FE caps. :

```
typedef struct
{
      FPPI_FE_Caps *fecaps; //Pointer to memory allocated for FE caps.
      uint32_t cbCorrelator; // Higher-level instance differentiator, opaque to the FP module.
} cm_fecaps_TranslatorContext_t;
```

Stores the translator's context related to getting the FE statistics:

```
typedef struct {
      npf_L3StatsEntry_t *stats_list; // memory allocated for statistics.
      uint32_t num_entries;  //number of ports
      uint32_t cbCorrelator;  //Higher-level instance differentiator, opaque to the FP module.
} cm_L3_stats_TranslatorContext_t;
```

3.8.5.2 Platform-Specific Component-Related Structures

Stores the user context related to getting the port number:

```
typedef struct
{
      cm_portnum_TranslatorContext_t *t_c;  //Translator context.
} cm_portnum_UserContext_t;
```

Stores the user context related to getting the FE caps:

```
typedef struct
{
      cm_fecaps_TranslatorContext_t *t_c;  //Translator context.
} cm_fecaps_UserContext_t;
```

Stores the user context related to getting the FE caps:

```
typedef struct {
      uint32_t *reqComp_Indctr; //Pointer to counter to keep track of translator request completion
      cm_L3_stats_TranslatorContext_t *t_c;  //Translator context.
      uint32_t port_index;  //Port index
      uint32_t portId;     //Id of the port.
} cm_L3_stats_UserContext_t;
```

Stores the pending user context info:

```
typedef struct {
      uint32_t type;  //Type of down-call
      uint32_t timestamp;  //Time stamp indicating the time of down-call
      void * u_c;  //pointer to the u_c
```

```
} cm_uc_info;
```

3.8.6 External API

All the external APIs, except the `fpm_cm_get_fe_caps()`, initialize and shutdown the APIs that are registered with the backend API

fpm_cm_mgr_get_fecaps()

Syntax

```
void fpm_cm_mgr_get_fecaps(bd_mgr_callback arg_Callback,char
*fename)
```

Description

This function extracts the `fename` information and invokes the platform-specific component `cm_get_fe_caps()` to get the FE capabilities. The B&D manager invokes this function.

The platform-specific function in turn invokes the stack driver API to get the required capabilities of the FE. Once the platform-specific component calls into the translator up-call, `(fpm_cm_mgr_get_fecaps_upcall)` the FE capability is sent to the B&D manager.

Parameters

`arg_Callback`	Callback function to pointer to call into Binding Discovery Manager when the FE capabilities are available
`fename`	Pointer to the Forwarding Element

Return Values

None

fpm_cm_mgr_get_fecaps_upcall()

Syntax

```
int32_t fpm_cm_mgr_get_fecaps_upcall(int32_t result,
cm_fecaps_TranslatorContext_t *t_c)
```

Description

This function is invoked by the platform-specific component in response to the FE capability query.

Parameters

`result`	Result of the down-call request, (-1 indicates failure)
`t_c`	Pointer to the translator context containing FE capabilities and the higher-level correlator

Return Values

- 0 - Success
- -1 - Failure

fpm_cm_mgr_set_L2_attributes()

Syntax

```
void fpm_cm_mgr_set_L2_attributes(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `cm_set_L2_attributes()`. This pointer is cast `npf_L2IntfAttrsList_t` structure defined in the `backendapi_types.h`.

Parameters

- cbcorrelator Callback Correlator, identifying instance of the call
- context Callback context
- response_data Pointer to packaged data from the backend API

Return Values

None

fpm_cm_mgr_set_L2_attributes_upcall()

Syntax

```
int32_t fpm_cm_mgr_set_L2_attributes_upcall(int32_t result,
cm_TranslatorContext_t *t_c)
```

Description

This function is invoked by the platform-specific component in response to set L2 attributes.

Parameters

- result Result of the down-call request, -1 means failure.
- t_c Pointer to the translator context containing higher-level correlator.

Return Values

- 0 - Success
- -1 - Failure

fpm_cm_mgr_set_L3_attributes ()

Syntax

```
void fpm_cm_mgr_set_L3_attributes(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component cm_set_L3_attributes.

When the IP address is configured on an interface, the CMM invokes the IPv4 Manager for adding the following routes:

1. Add subnet route with L2 Index = 0
2. Add directed broadcast route
3. Add exact match route

Parameters

- Cbcorrelator — Callback correlator, identifying instance of the call.
- Context — Callback context.
- response_data — Pointer to packaged data from the backend API containing L3 Attribute information (`npf_L3IntfAttrsList_t`)

Return Values

None

fpm_cm_mgr_set_L3_attributes_upcall()

Syntax

```
int32_t fpm_cm_mgr_set_L3_attributes_upcall(int32_t result,
cm_TranslatorContext_t *t_c)
```

Description

This function is invoked by the platform-specific component in response to set L3 attributes.

Parameters

- Result — Result of the down-call request, -1 means failure
- t_c — Pointer to the translator context containing higher-level correlator

Return Values

- 0 - Success
- -1 - Failure

fpm_cm_mgr_delete_L3_attributes ()

Syntax

```
void fpm_cm_mgr_set_L3_attributes(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description:

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `cm_delete_L3_attributes()`.

Parameters

- cbcorrelator — Callback correlator, identifying instance of the call.
- context — Callback context.
- response_data — Pointer to packaged data from the backend API containing L3 attribute information (`npf_L3IntfAttrsList_t`).

Return Values

None

fpm_cm_mgr_delete_L3_attributes_upcall()

Syntax

```
int32_t fpm_cm_mgr_delete_L3_attributes_upcall(int32_t result,
cm_TranslatorContext_t *t_c)
```

Description

This function is invoked by the platform-specific component in response to delete L3 attributes.

Parameters

- result Result of the down-call request, -1 means failure.
- t_c Pointer to the translator context containing higher-level correlator.

Return Values

- 0 - Success
- -1 - Failure

fpm_cm_mgr_get_L3_stats ()

Syntax

```
void fpm_cm_mgr_get_L3_stats(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `cm_get_L3_stats()`.

Parameters

- cbcorrelator Callback correlator, identifying instance of the call.
- context Callback context.
- response_data Pointer to packaged data from the backend API containing port ID information (`npf_PortList_t`).

Return Values

None

fpm_cm_mgr_get_L3_stats_upcall()

Syntax

```
int32_t fpm_cm_mgr_get_L3_stats _upcall(int32_t result,
cm_L3_stats_TranslatorContext_t *t_c)
```

Description

This function is invoked by the platform-specific component in response to L3 stats query.

Parameters

- `result` Result of the down-call request, -1 means failure.
- `t_c` Pointer to the translator context containing higher-level correlator and L3 statistics.

Return Values

- 0 - Success
- -1 – Failure

3.8.7 Platform-Specific Component APIs

The stack driver core component helps in configuration and management of the interfaces. The statistics information is managed by the respective TX/RX core components.

Most of the calls to the IXA SDK 3.51 are asynchronous in behavior and the result of the request is available in the callback from the IXA SDK 3.51.

Table 3. Config manager platform-specific component to IXA SDK 3.51 mapping table

Platform-Specific Function	Notes	Core Component	IXA SDK 3.51 API
cm_get_fecaps()	This call invokes two calls into the IXA SDK 3.0. One to get the number of ports, and then to get the properties for all the ports. Once the properties are available, the CM manager's respective up-call is invoked.	Stack Driver	ix_cc_stkdrv_aync_get_num_ports() ix_cc_stkdrv_aync_get_property()
cm_get_l3_stats()	Invokes two calls one to Rx CC to get the RX Stats, and one to FP CC to get the Tx side stats. The FP CC invokes calls into the egress side and calls back once the stats are available.	Ethernet Interface RX FP CC	ix_cc_eth_rx_async_get_statistics_info – ix_cc_fpm_async_get_statistics_info(). This call invokes the egress API via the egress FP CC.
cm_set_L2_attributes()	Invokes call to set the link speed, MTU, and so on., by appropriately filling out the property ID and the properties structure	SDK properties	ix_cc_async_set_property
cm_set_L3_attributes	Invokes call to set IP addresses, by appropriately filling out the property ID and the properties structure	SDK Properties	ix_cc_async_set_property

cm_delete_L3_attributes()	Invokes call to set IP addresses, by appropriately filling out the property ID and the properties structure	SDK Properties	ix_cc_async_set_property

The following sections describe the implementation details of both the translator and platform-specific component for CM manager for getting statistics. The other APIs are implemented in a similar way.

3.8.8 Setting and Deleting properties

The calls into the stack driver properties API to set and delete properties are synchronous and the status is reported in the context of the backend API down call. The calls to get the properties, such as getting port number or FE capabilities, are asynchronous and status is reported to the CP in the context of the IXA SDK 3.51 callback.

3.8.9 Getting Statistics

Only `fpm_cm_mgr_get_L3_statistics()` is supported. To get these statistics, Ethernet RX and TX external messaging APIs have to be invoked. Invoking the Ethernet RX core component gets the RX statistics. As the Ethernet TX resides on the egress side, the CM manager invokes the FP CC on the ingress side. The ingress FP CC sends message to the FP CC in the egress side that in turn calls into the ethernet TX CC for statistics information. Following sequence diagram and pseudo code helps to understand the implementation details for statistics gathering for two ports.

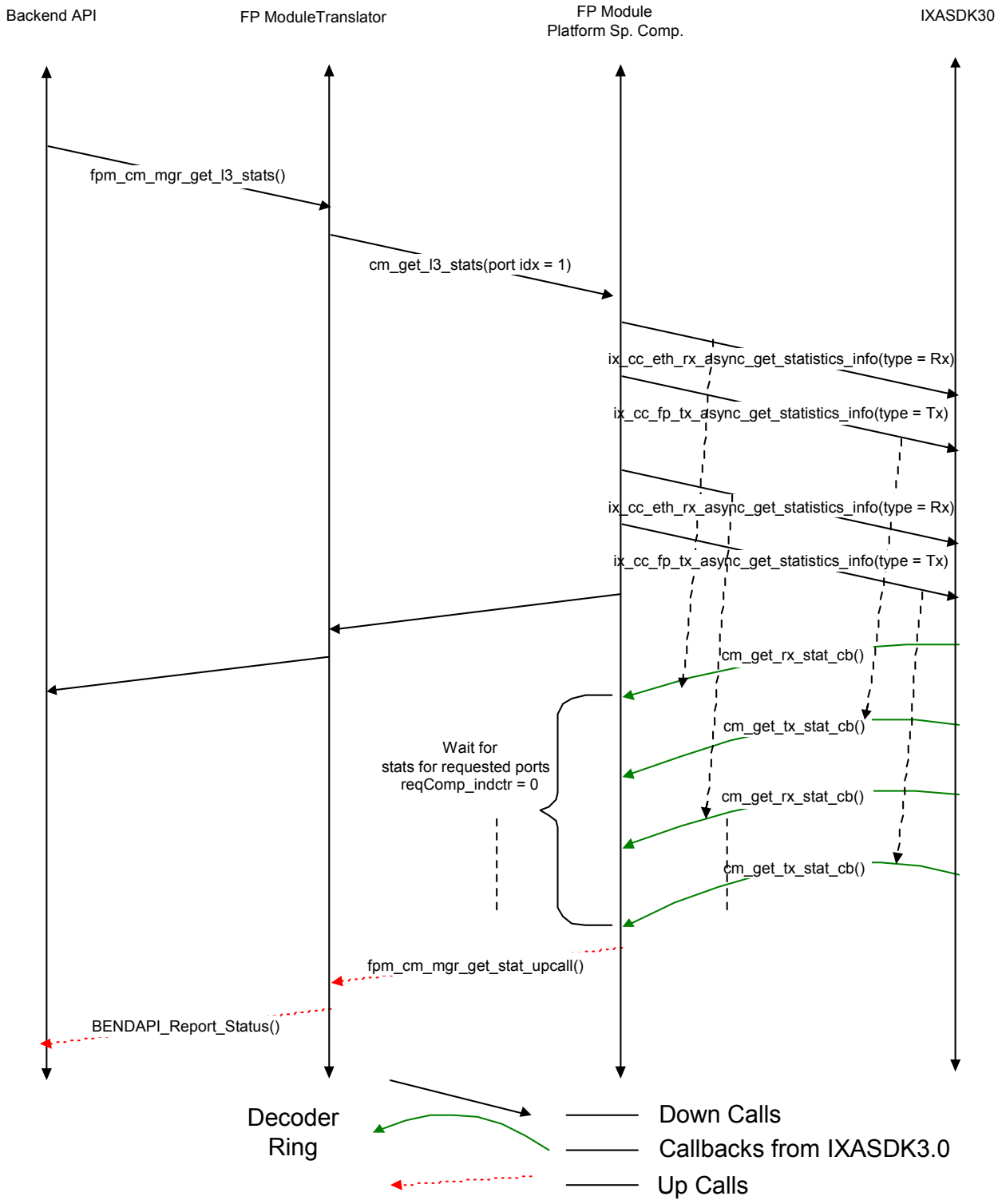


Figure 11: Sequence diagram for getting Ethernet statistics for two ports

```
fpm_cm_mgr_get_L3_statistics(cbCorrelator)
{
      num_ports = 2;
- allocate memory for translator context (t_c)
- fill out the fields of t_c
t_c->stat_list = memory to hold stat into for num_ports
t_c->cbCorrelator  = cbCorrelator;

- call platform-specific component
cm_get_L3_statistics(t_c, num_ports);
}
cm_get_L3_statistics(t_c, num_ports)
{
      - allocate memory for reqComp_Indctr
      - set reqComp_Indctr to num_ports, i.e. 2
for(i = 0;i < num_ports; ++i)
{
for (j = 0; j < 2; ++j)
{
- allocate memory for user context (u_c)
- fill out this u_c fields
u_c->reqComp_indctr = reqComp_Indctr;
u_c->t_c = t_c
u_c->port_index = i;
- call into the IXA SDK 3.5
if(j == 0) //to get RX stats
{
ix_cc_eth_rx_async_get_statistics_info(u_c,
cm_L3_stat_callback_rx);
}
if(j == 1)//to get TX stats
{
ix_cc_fpm_rx_async_get_statistics_info(u_c,
cm_L3_stat_callback_tx);
}
}
}
}
cm_L3_stat_callback_rx(result, u_c, pBuffer)
{
      //check if all calls have completed
if (*u_c-> reqComp_indctr != 0)
{
*u_c-> reqComp_indctr--;
//copy statistics information for the correct port_index
```

```
copy_rx_info(u_c->t_c->stat_list, pBuffer)
}
if (*u_c-> reqComp_indctr == 0)
{
fpm_cm_mgr_L3_stat_upcall(t_c);
free_counter();
}
ree_this_user_context_resources();
}

cm_L3_stat_callback_tx(result, user_context, pBuffer)
{
      //check if all calls have completed
if (*u_c-> reqComp_indctr != 0)
{
*user_context-> reqComp_indctr--;
//copy statistics information for the correct port_index
copy_tx_info(u_c->t_c->stat_list, pBuffer)
}
if (*u_c-> reqComp_indctr == 0)
{
fpm_cm_mgr_L3_stat_upcall(t_c);
free_counter();
}
free_this_user_context_resources();
}
fpm_cm_mgr_L3_stat_upcall(t_c)
{
Call BackendAPI_Status_Report(t_c->cbCorrelator,
t_c->stat_list)};
free_stat_info();
}
```

3.8.10 Getting Port Properties

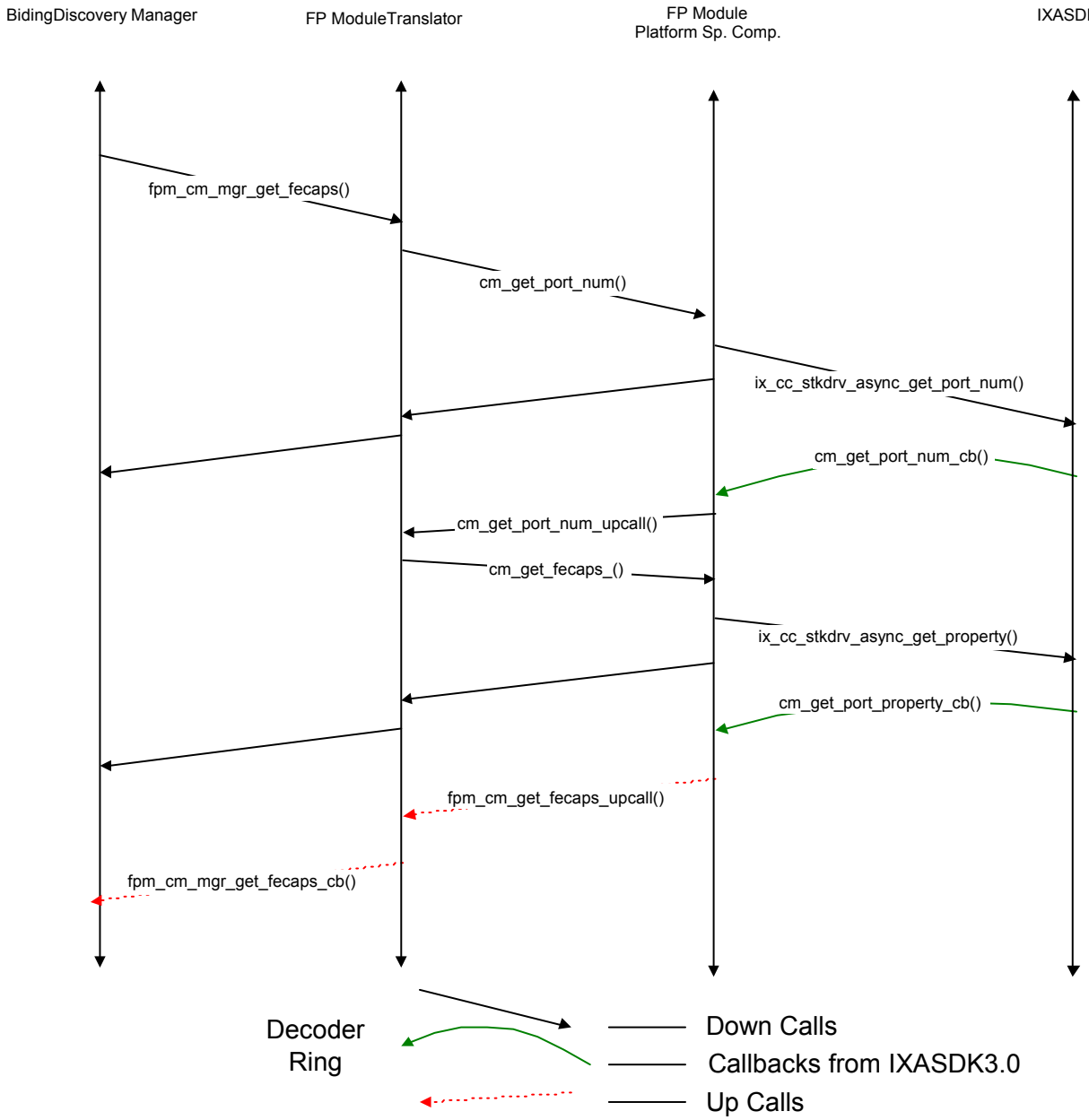


Figure 12: Sequence diagram for getting port properties

The B&D manager calls into the CM manager for FE capabilities during the FE bind operation. Figure 12: Sequence diagram for getting port properties depicts the sequence of events. As a first step the number of ports is requested from the IXA SDK 3.51. This information is available on the callback from the stack driver CC. The number of ports is made available to the CMM translator via the up-call. The translator now builds the required translator context that comprises the allocation for the FE caps for this number of ports, and invokes into the platform-specific component. The translator up-call is invoked once the properties for all the ports are available. The

CMM translator calls back the binding and discovery manager with the requested information on FE capabilities.

3.9 IPv4 Manager

This section gives an overview of the IPv4 FP plug-in manager.

3.9.1 Functionality

The IPv4 FP plug-in manager maps IPv4 NPF calls to IPv4 and Ethernet RX core components, via the ingress FP core component, for route and ARP table management respectively.

The IPv4 FP plug-in manager's translator registers callback functions with the backend API for the various services like adding/deleting routes, and next hops . The platform-specific component interacts with the IXA SDK 3.51 API.

3.9.2 Execution Context

All the IPv4 manager down-calls execute in the context of the backend API callback functions registered during the initialization time. In the cases where the calls into the IXA SDK 3.51 are synchronous, the up-calls execute in the context of backend API callback functions, such as, `fpm_ipv4_mgr_flush_prefixes()`. If the calls into the IXA SDK 3.51 are asynchronous then up-calls execute in the context of the callbacks from the IXA SDK 3.51, for example, `fpm_ipv4_mgr_add_prefix()`.

3.9.3 Initialization

The FP boot manager invokes the initialization function `fpm_ipv4_mgr_init ()`. This function registers callbacks with the backend API for setting up route tables in the IPv4 CC and invokes the platform-specific initialization function.

3.9.4 Shutdown

The FP boot manager invokes the shutdown function `fpm_ipv4_mgr_shutdown ()`. This function de-registers the callbacks from the backend API and invokes the platform-specific shutdown function.

3.9.5 Data Structures

This section provides information on the data structures used by the IPv4 FP plug-in manager.

3.9.5.1 IPv4 Translator-related Structures

```
typedef struct {
    NPF_RET            retStatus;
    npf_AddressType_t  type;
    union
    {
        NPF_IPv4Address_t        ipv4Addr;
        NPF_MediaAddressEntry_t  l2Addr;
    } u;
} npf_IPv4_Status_t;
typedef struct
{
npf_IPv4_Status_t *status; // Pointer to structures containing
status info. for each //add/delete/update request.
      uint32_t num_entries; // number of add/delete/update entries
      uint32_t cbType; //Type of down-call request
      uint32_t cbCorrelator; // Higher-level instance
differentiator, opaque to the FP module.
} ipv4_TranslatorContext_t;
```

3.9.5.2 IPv4 Platform Component-related Structures

```
typedef struct
{
      uint32_t *reqComp_Indctr; // Pointer to counter to keep
track of translator request completion
      ipv4_TranslatorContext_t *t_c; // Translator context.
      uint32_t index; //index indicating the offset of the this
u_c
} ipv4_UserContext_t;
```

3.9.6 External API

All the external APIs are registered with the backend API except the initialize and shutdown APIs that are called by the FP boot manager.

fpm_ipv4_mgr_upcall()

Syntax

```
int32_t fpm_ipv4_mgr_upcall(int32_t result,
ipv4_TranslatorContext_t *t_c)
```

Description

This function is invoked by the platform-specific components in response to adding/deleting/updating prefixes, next hops or ARPs.

Parameters

- result Result of the down-call request, -1 means failure.
- t_c Pointer to the translator context containing the higher-level correlator and status of adding next hops.

Return Values

- 0 - Success
- -1 - Failure

fpm_ipv4_mgr_add_next_hop()

Syntax

```
void fpm_ipv4_mgr_add_next_hop(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ipv4_add_next_hop()`.

The platform-specific function in turn adds entries to the next hop table and L2 table. To indicate the completion of the request the platform-specific component calls into the translator up-call (`fpm_ipv4_mgr_upcall`), and status of adding next hops is sent to the backend module.

Parameters

- cbcorrelator Callback correlator, identifying instance of the call.
- context Callback context.
- response_data Pointer to packaged data from the backend API containing next hop information.

Return Values

None

fpm_ipv4_mgr_delete_next_hop ()

Syntax

```
void fpm_ipv4_mgr_delete_next_hop(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ipv4_delete_next_hop().`

The platform-specific function in turn deletes entries in the next hop table and L2 table. To indicate the completion of the request the platform-specific component calls into the translator up-call (`fpm_ipv4_mgr_delete_next_hop_upcall`), and status of deleting next hops is sent to the backend module.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call.
- `context` Callback context.

- `response_data` Pointer to packaged data from the backend API containing next-hop information.

fpm_ipv4_mgr_add_prefix ()

Syntax

```
void fpm_ipv4_mgr_add_prefix(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ipv4_add_prefix().`

The platform-specific function in turn adds entries in the prefix table in the ingress. To indicate the completion of the request, the platform-specific component calls into the translator up-call and status of adding prefixes is sent to the backend module.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` Pointer to packaged data from the backend APIcontaining prefix information.

Return Values

None

fpm_ipv4_mgr_update_prefix ()

Syntax

```
void fpm_ipv4_mgr_update_prefix(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ipv4_update_prefix().`

The platform-specific function in turn updates entries to the prefix table in the ingress for the existing `NHID`. To indicate the completion of the request, the platform-specific component calls into the translator up-call, and status of update is sent to the backend module.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` Pointer to packaged data from the backend API containing the prefix information.

Return Values

None

fpm_ipv4_mgr_delete_prefix ()

Syntax

```
void fpm_ipv4_mgr_delete_prefix (FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ipv4_delete_prefix()`. This pointer is cast to next-hop structure defined in the `backendapi_types.h`.

The platform-specific function in turn deletes entries in the prefix table in the ingress. To indicate the completion of the request, the platform-specific component calls into the translator up-call `(fpm_ipv4_mgr_upcall)`, and status of delete is sent to the backend module.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` Pointer to packaged data from the backend API containing prefix information.

Return Values

None

fpm_ipv4_mgr_flush_prefixes ()

Syntax

```
void fpm_ipv4_mgr_flush_prefixes(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function invokes the platform-specific component `ipv4_flush_prefixes()`. The platform-specific function, in turn, deletes all routes and next-hops from the prefix and next-hop tables.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` NULL.

Return Values

None

fpm_ipv4_mgr_flush_next_hops ()

Syntax

```
void fpm_ipv4_mgr_flush_prefixes(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function invokes the platform-specific component `ipv4_flush_prefixes()`. The platform-specific function, in turn, deletes all routes and next-hops from the prefix and next-hop tables.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` NULL.

Return Values

None

fpm_ipv4_mgr_add_arp ()

Syntax

```
void fpm_ipv4_mgr_add_arp (FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the response_data pointer and invokes the platform-specific component `ipv4_add_arp().`

The platform-specific function in turn adds entries to the ARP table on the egress. To indicate the completion of the request, the platform-specific component calls into the translator up-call (`fpm_ipv4_mgr_upcall`), and status of adding ARP is sent to the backend module.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` Pointer to packaged data from the backend API containing ARP information.

Return Values

None

fpm_ipv4_mgr_delete_arp ()

Syntax

```
void fpm_ipv4_mgr_delete_arp (FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ipv4_delete_arp().` This pointer is cast to next hop structure defined in the `backendapi_types.h`.

The platform-specific function in turn deletes entries in the ARP table in the ingress. The platform-specific function in turn adds entries to the ARP table on the egress. To indicate the completion of the request, the platform-specific component calls into the translator up-call (`fpm_ipv4_mgr_delete_arp_upcall`), and status of delete is sent to the backend module.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` Pointer to packaged data from the backend API.

Return Values

None

fpm_ipv4_mgr_flush_arps ()

Syntax

```
void fpm_ipv4_mgr_flush_arps(FPPI_CBCORRELATOR cbcorrelator,
FPPI_CNTEXT context, void *response_data)
```

Description

This function invokes the platform-specific component `ipv4_flush_arps()`. The platform-specific function in turn deletes all routes and next hops from the ARP tables.

Parameters

- `cbcorrelator` Callback correlator, identifying instance of the call.
- `context` Callback context.
- `response_data` NULL.

Return Values

None

3.9.7 Platform-Specific Component APIs

The IPv4 core component helps maintain three tables that are consulted for every incoming packet. These three tables are:

1. Prefix table (PT): This table resides on the ingress processor of the IXP nNetwork processor. The entries are indexed by prefixes and the look-up yields Next-hop ID (NHID) passed down from the client stacks through NPF APIs.
2. Next-hop table (NHT): This table resides on the ingress processor of the IXP network processor. The entries are indexed by NHID and yield the L2 ID and the blade IDs in addition to other parameters.
3. L2 table (L2T): This table resides on the egress processor of the IXP network processor. The entries are indexed by L2 IDs and yield the L2 encapsulation header for the outgoing packet.

There are three lookups for every incoming packet traversing the IXP2000 series network processor. In brief, when a packet comes into the router the first (PT) lookup gives the NHID and then the second (NHT) lookup gives the blade ID and the L2 ID for that blade; for Ethernet based media type this blade ID information is used for a third (L2T) lookup on the egress to get the layer-2 encapsulation and the packet is eventually sent out. In case of POS media type, there is no L2T lookup. The L2 table management is done via the FP CC API. Whereas the L2T is specific to a blade, the exact same PT and NHT are maintained in all the blades of a multi-blade router.

Most of the calls to the IXA SDK 3.51 are asynchronous in behavior and the result of the request is available in the callback from the IXA SDK 3.51. The FP module communicates with the egress side via the FP CC. Table 4 gives the platform-specific component API to IXA SDK 3.51 API mapping.

Table 4. IPv4 platform-specific component API to IXA SDK 3.51 API mapping table

Platform-Specific Function	Notes	Core Components	IXA SDK 3.51 API
ipv4_add_next_hop()	Invokes calls to populate the Next hop table and in case of Ethernet Tx/Rx interfaces the L2 table is also populated. IXA SDK supports updating next hop; in case of Ethernet the existing L2 entry is deleted before the newer L2 entry is added.	IPv4 FP CC IPv4 FP CC FP CC	ix_cc_ipv4_async_add_next_hop() ix_cc_fpm_sync_add_L2_entry() ix_cc_ipv4_async_update_next_hop() ix_cc_fpm_sync_del_L2_entry() ix_cc_fpm_sync_add_L2_entry()
ipv4_delete_next_hop()	Invokes calls to delete the next hop table and in case of Ethernet, the corresponding L2 entry via the FP CC API.	IPv4 FP CC	ix_cc_ipv4_async_delete_next_hop() ix_cc_fpm_sync_del_L2_entry()
ipv4_flush_next_hops()	Invokes calls to purge the route and Next hop tables and the L2 table.	IPv4	ix_cc_ipv4_async_purge_rtm() ix_cc_fpm_async_purge_L2_table()
ipv4_add_prefix()	Invokes calls to add route entry in the route table.	IPv4	ix_cc_ipv4_async_add_route()
ipv4_update_prefix()	Invokes calls to update route entry in the route table.	IPv4	ix_cc_ipv4_async_update_route()
ipv4_delete_prefix()	Invokes calls to delete route entry in the route table.	IPv4	ix_cc_ipv4_async_delete_route()
ipv4_flush_prefixes()	Invokes calls to purge the route table.	IPv4	ix_cc_ipv4_async_purge_routes()

ipv4_add_arp()	Invokes calls to add ARP entry in the ARP table.	FP CC	ix_cc_fpm_async_add_arp()
ipv4_delete_arp()	Invokes calls to delete ARP entry in the ARP table.	FP CC	ix_cc_fpm_async_delete_arp()
ipv4_purge_arp()	Invokes calls to purge ARP table.	FP CC	ix_cc_fpm_async_purge_arp()

The following sections give the implementation details for adding routes and next hops. The other APIs (like adding/deleting ARPs, purging routes etc) are implemented in a similar way.

Figure 13 shows NPF to IXA SDK 3.51 mapping for adding next hops and routes. As can be seen, when the `fpm_add_next_hop()` comes along, the IPv4 manager makes two calls to the IXA SDK 3.51; one to add entry to the next hop table on the ingress and the other to populate the L2 table on the egress side. The NHIDs in the figure are passed from the NPF compliant client stacks. The dotted lines in red in Figure 13 show the calls related to updating the next hop for an existing NHID.

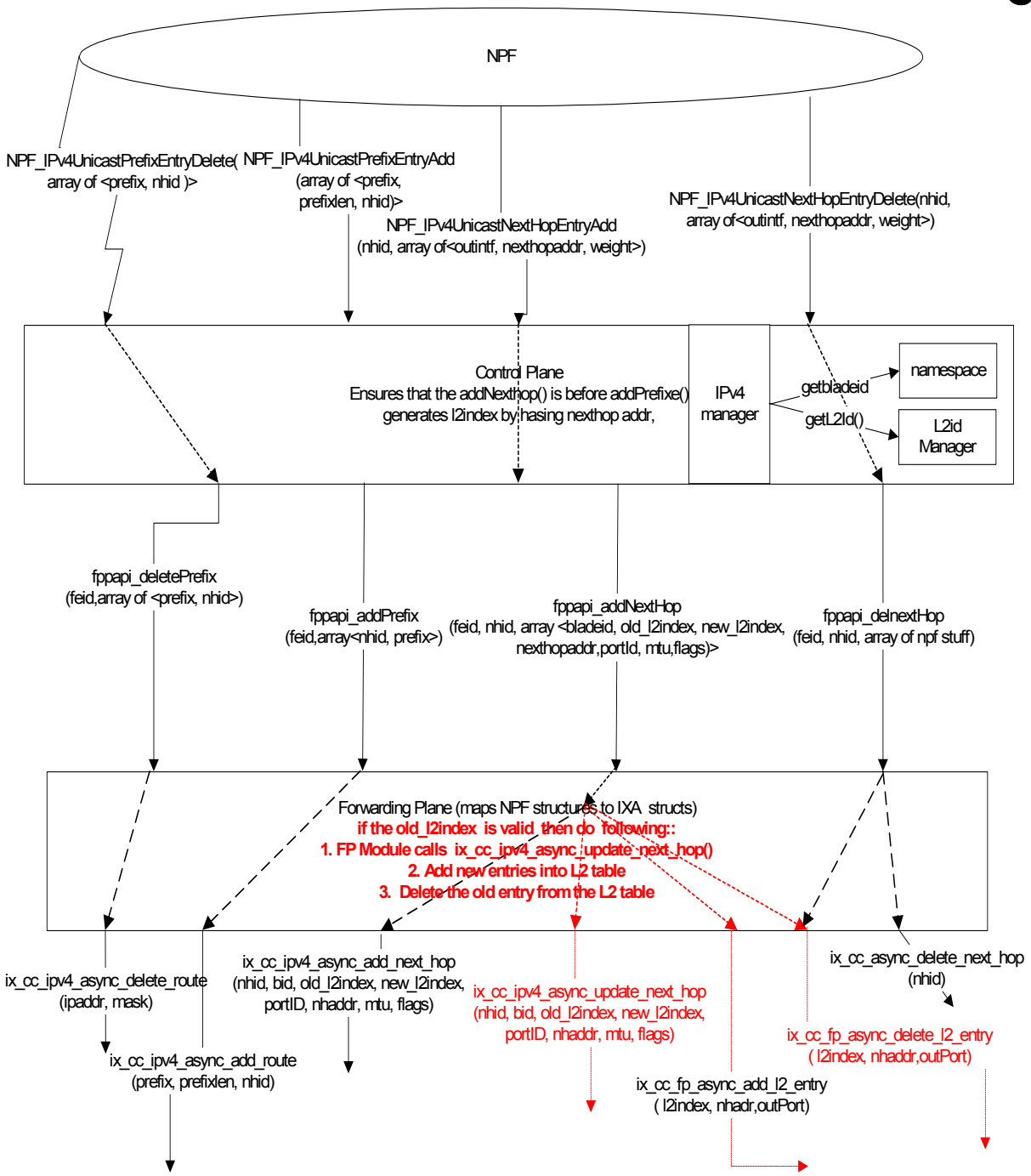


Figure 13: NPF-FP module-IXA SDK 3.51 mapping

3.9.8 Adding Routes and Next Hops

Adding a route assumes that next-hop is previously added for the particular NHIDs. For more details on this, refer to IPv4 module in the CP [4]. As the calls into the IXA SDK 3.51 are

asynchronous, the status of each call is indicated in the respective callback function in platform-specific component. When all callbacks for the respective down-calls have arrived, the translator up-call is invoked and the cumulative status is given to the backend API via the `ipv4_Status_t structure`. The sequence diagram in Figure 14 and pseudo code helps to understand the implementation details for adding two routes for an existing next-hop.

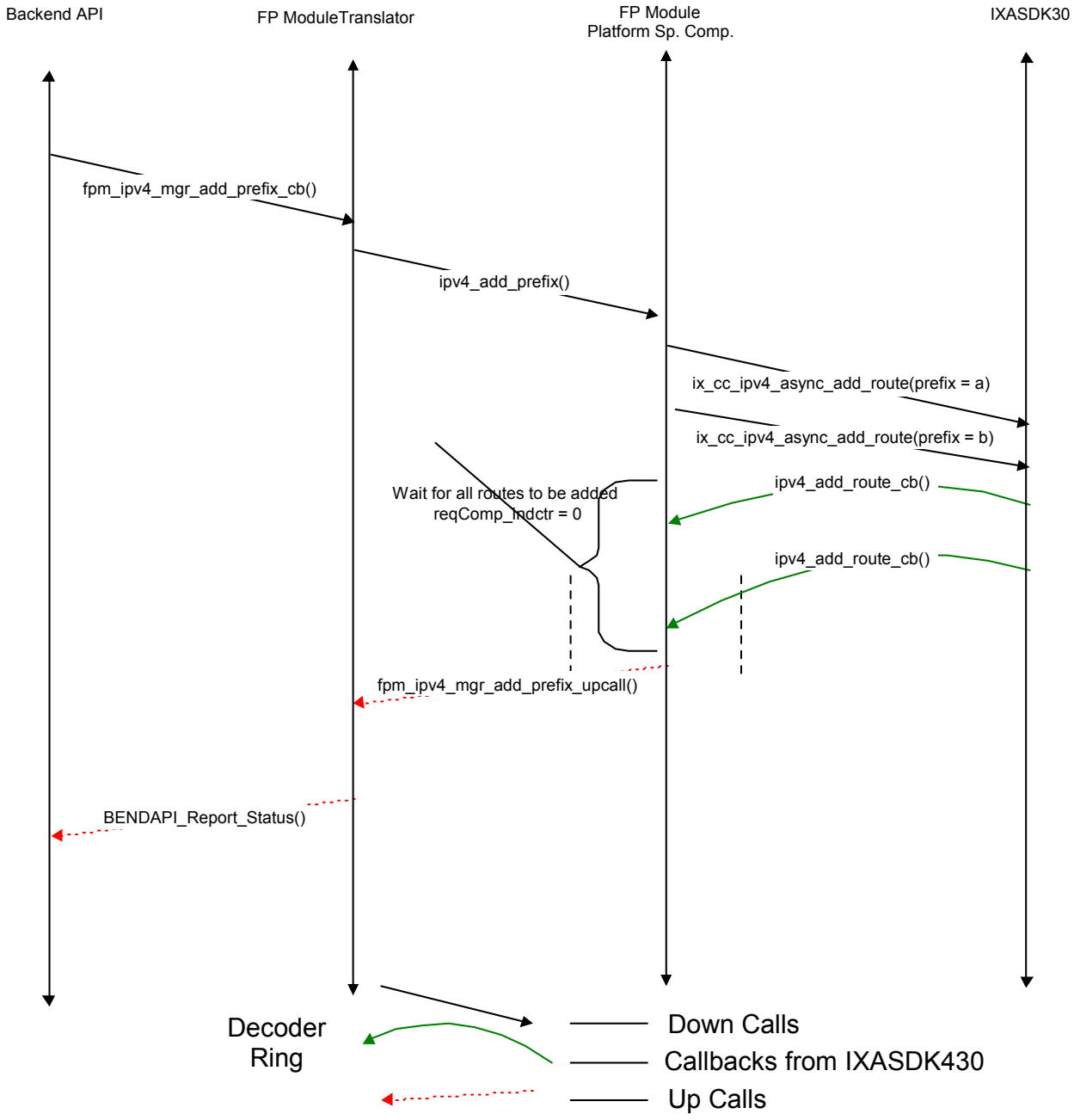


Figure 14: Sequence diagram for adding a route

```
fpm_ipv4_mgr_add_prefix(cbCorrelator, route_info)
{
      num_routes = 2;
      - allocate memory for the translator context (t_c)
      - Allocate memory to hold status info for num_routes
- Fill the t_c fields
      for ( num_routes)
{
t_c->status[i].type = NPF_IPv4_ADDRESS;
t_c->status[i]. u.ipv4Addr = route_info[i].addr;
t_c->cbCorrelator = cbCorrelator;
}
- call platform-specific component.
ipv4_add_prefix(translator_context, route_info, num_routes);
}
ipv4_add_prefix(t_c, route_info, num_routes)
{
      -  allocate memory for reqComp_Indctr
      - set reqComp_Indctr to num_routes, i.e. 2
for(i=0;i<num_routes;++i)
{
      - allocate memory for user context (u_c)
      - Fill u_c fields
u_c->reqComp_Indctr = reqComp_Indctr;
u_c->translator_context = translator_context;
u_c->index = i;
- call IXA SDK 3.5
ix_cc_ipv4_async_add_route(u_c, route_info,
ipv4_add_route_callback());
}
}
ipv4_add_route_callback(result, user_context)
{
      uint32_t indctr = *u_c->reqComp_indctr;
//check to see if all the callbacks have arrived
if (indctr != 0)
      {
- decrement reqComp_indctr(*u_c->reqComp_indctr--)
Set the error status for this request
u_c->t_c->status[u_c->index].retStatus = result;
}
if (indctr == 0)
      {
- call the translator upcall
fpm_ipv4_mgr_add_route_upcall(translator_context);
free_counter;
```

```
}
free_this_user_context_resources();
}
fpm_ipv4_mgr_add_route_upcall(t_c)
{
Call BackendAPI_Status_Report(translator_context->cbCorrelator,
t_c->status);
free_status_memory();
free routeinfo()
}
```

NPF defines an efficient way of updating the Next-hop. As per this definition there is no need to re-download the prefixes if a next-hop changes. Refer to CP IPv4 module document [4] for more information. As the L2T is specific to the blade, the new next-hop information might imply L2T update of another blade and the L2T entry in the original blade needs to be deleted. The flow chart in Figure 15 gives the algorithm to ensure proper synchronization of the L2T in all the blades.

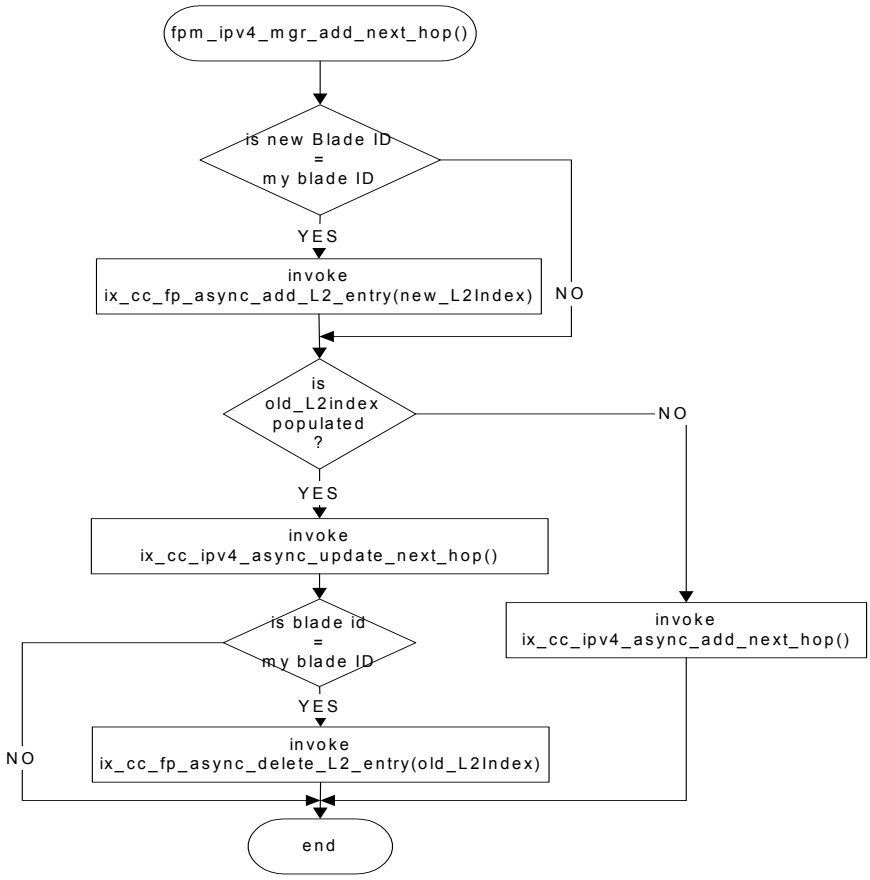


Figure 15: Algorithm to manage the L2 table for adding Next-hop

3.9.9 Adding/Deleting ARP Entries

The ARP table is maintained in the egress side of the FE. So, when the add ARP entry request comes to the IPv4 manager, it invokes the ingress FP core component (FP CC) via the `ix_cc_fp_async_add_arp()` . The FP CC is responsible for communicating with the egress FP CC, which in-turn invokes the TX Ethernet CC. On Completion of the request, the ingress FP CC calls back the IPv4 manager's platform-specific component. The status reporting to the backend API is done in the similar way as done for add route status reporting. The type field in the `ipv4_Status_t` is set to indicate MAC address.

3.10 Event Manager

This is responsible for propagating events, such as, port-up and port-down, to the transport plug-in through the backend API's event notification interface.

3.10.1 Functionality

The event manager handles events arising from the underlying NPU. It currently supports the link up/ link down events. It generates a backend API event notification whenever it detects a change in the Link State.

3.10.2 Execution Context

The event handler thread executes in its own context, sending event notification upwards through the backend API event notification mechanism.

3.10.3 Initialization

On initialization, the event handling thread is created.

3.10.4 Shutdown

On shutdown, the event handler thread is destroyed.

3.10.5 IXA SDK 3.51 Implementation

The linkstate module determines the state of the IXP interface. It receives an event from the ingress FP CC on the link status change.

3.11 ATM Manager

This section gives an overview of the ATM FP plug-in manager.

3.11.1 Functionality

The ATM FP plug-in manager maps the ATM NPF calls to ATM core components for creating/deleting/getting statistics of Port/VCC/AAL2 channel.

The ATM FP plug-in manager's translator registers the callback functions with the backend API for various services like adding/deleting Port, VCC, AAL2 channel and so on. The platform-specific component interacts with the IXA SDK 3.51 API.

3.11.2 Execution Context

All the ATM manager down-calls execute in the context of the backend API callback functions registered during the initialization time. All the ATM manager up-calls are asynchronous and they execute in the context of the callbacks from the IXA SDK 3.51.

3.11.3 Initialization

The FP boot manager invokes the initialization function `fpm_atm_conn_mgr_init ()`. This function registers the callbacks with the backend API for setting/deleting/getting statistics of Port/VCC in the ATM CC and invokes the platform-specific initialization function.

3.11.4 Shutdown

The FP boot manager invokes the shutdown function `fpm_atm_mgr_shutdown ()`. This function de-registers the callbacks from the backend API and invokes the platform-specific shutdown function.

3.11.5 Data Structures

This section provides information on the data structures used by the ATM FP plug-in manager.

3.11.5.1 ATM Platform-Specific Data Structures

For the IXA_SDK_3.1 ATM SAR CC platform, the ATM translator maintains three hash lists. One each for ports and VCCs created. If the AAL type of the VCC connection is AAL2, then the VCC hash node also has an AAL2 Hash List.

One of the main reasons for maintaining this hash list is to maintain the mapping between the port/VCC/AAL2 Id provided by the CP-PDK user and the corresponding port/VCC/AAL2 handle provided by the `IXA_SDK_3.1 ATMSAR CC`.

The following data structures are used for maintaining the hash list and the correlators used for communicating with the IX_SDK_3.1 ATMSAR CC.

```
typedef struct
{
      NPF_VccAddr_t     conection;        /* VPI/VCI */
      NPF_uint32_t      portId;           /* Port Id */
      NPF_IfHandle_t    upperIfHandle;    /* Upper Interface
Handle */
      NPF_IfAAL_t       aalType;          /* AAL Type */
      PdkHashCb         aal2ChanIdHashList;/* AAL 2 Hash List
Control Block */
} limePointAtmVccHashInfo_t;

typedef struct
{
      NPF_uint16_t      aal2Cid;          /* AAL2 Channel Id */
      NPF_uint16_t      aal2Handle;       /* AAL2 Handle */
}limePointAtmAal2HashInfo_t;

typedef struct
{
      NPF_uint32_t      portId;           /* Port Id */
      NPF_uint16_t      portHandle;       /* Port Handle */
      DList             vccHashInfoList;  /* Linked List of VCCs
mapped to this port */
}limePointAtmPortHashInfo_t;

tyepdef struct
{
      NPF_uint32_t            portId;     /* Port Id */
      FPPI_CBCORRELATOR       corr;       /*  Correlator */
}limePointPortCorrelator_t;

typedef struct
{
      NPF_VccAddr_t           connection; /* VPI/VCI */
      NPF_uint32_t            portId;     /* Port Id */
      FPPI_CBCORRELATOR       corr;       /* Correlator */
      NPF_IfAAL_t             aalType;    /* AAL Type */
}limePointVccCorrelator_t;

tyepdef struct
{
      NFP_VccAddr_t           connection; /* VPI/VCI */
```

```
      NPF_uint32_t            portId;     /* Port Id */
      NPF_uint16_t            aal2Cid;    /* AAL2 Channel Id */
      FPPI_CBCORRELATOR       corr;       /* Correlator */
}limePointAal2Correlator_t;
```

3.11.6 External API

All the external APIs are registered with the backend API, except the initialize and shutdown APIs that are called by the FP boot manager.

fpmIfATM_IfAttrSetFunc()

Syntax

```
Void fpmIfATM_IfAttrSetFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_port_create()`.

On a dual processor system like Angel Island, the CP-PDK runs on the ingress, and the ATMSAR main CC resides on the egress processor. This function internally sends the received port attribute information to the egress, where the platform-`specific ix_cc_atmsar_async_port_create` function is called.

The platform-specific function in turn creates the ATM port with the port speed attributes passed to it. To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointPortCreateCBFunc`. In the callback function, if the status of the port create operation is successful, then the ATM translator adds the port information in its port hash list. Each node in the port hash list contains the following information:

- Port Id
- Port Handle, which is the handle returned by the ATM SAR CC
- List of ATM VCCs mapped to this port. When the port is deleted, this information is needed to delete all the VCCs associated with this port.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to packaged data from the backend API containing the ATM port attributes information

Return Values

None

fpmIfATM_IfDeleteFunc()

Syntax

```
Void fpmIfATM_IfDeleteFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_port_remove()`.

Currently on a dual processor system like Angel Island, the CP-PDK runs on the ingress, and the ATMSAR main CC resides on the egress processor. This function retrieves the port handle stored in ATM translators port hash list and then sends this port handle information to the egress, where the platform-specific `ix_cc_atmsar_async_port_remove` function is called.

The platform-specific function removes the ATM port. To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointPortDeleteCBFunc`. In the callback function, if the status of the portdelete operation is success, then the ATM translator removes the port information from its port hash list. Then it calls the platform-specific component `ix_cc_atmsar_asyn_vc_remove` function to delete all the VCCs mapped to this port.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to the packaged data from the backend API containing the ATM port ID information

Return Values

None

fpmIfATM_IfGenStatsGetFunc()

Syntax

```
Void fpmIfATM_IfGenStatsGetFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_get_port_stats()`.

On a dual processor system like Angel Island, the CP-PDK runs on the ingress, and the ATMSAR main CC resides on the egress processor. This function retrieves the port handle stored in the ATM translators port hash list and then sends this port handle information to the egress where the platform-specific `ix_cc_atmsar_async_get_port_stats` function is called.

The platform-specific function in turn gets the statistics of the ATM port. To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointPortStatsGetCBFunc`.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call

- `context` Callback context
- `response_data` Pointer to the packaged data from the backend API containing the ATM port ID information

Return Values

None

fpmIfATM_VccSetFunc()

Syntax

```
Void fpmIfATM_VccSetFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_vc_create()`.

On a dual processor system like Angel Island, the CP-PDK runs on the ingress, and the ATMSAR main CC resides on the egress processor. This function internally sends the received VCC attribute information to the egress, where the platform-specific `ix_cc_atmsar_async_vc_create` function is called.

The platform-specific function in turn creates the ATM VCC with the attributes passed to it. To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointVccCreateCBFunc`. In the callback function, if the status of the VCC create operation is success, then the ATM translator adds the VCC information in its VCC hash list. Each node in the VCC hash list contains the following information:

- VPI/VCI value
- Port Id
- VCC handle which is the handle returned by ATM SAR CC
- Upper interface handle used for slow path packets. This is needed to inform the layer 3 of the layer-3 interface from which the packet was received.
- AAL type
- Hash list of AAL2 channels mapped to this VCC. This field is used only for AAL2 connections.

The ATM translator also adds the information of the newly created VCC in the VCC list of the port on which the VCC is mapped.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to the packaged data from the backend API containing the ATM VCC attributes information

Return Values

None

fpmIfATM_VccBindFunc()

Syntax

```
Void fpmIfATM_VccBindFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and stores the layer-3 interface handle in the Hash Node of the associated VCC in the VCC hash List.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to the packaged data from the backend API containing the ATM VCC information and the Layer-3 interface handle

Return Values

None

fpmIfATM_VccDelFunc()

Syntax

```
Void fpmIfATM_VccDelFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_vc_remove()`.

On a dual processor system like Angel Island, the CP-PDK runs on the ingress whereas the ATMSAR Main CC resides on the egress processor. This function retrieves the VCC handle stored in ATM translators VCC hash List and then sends this VCC handle information to the egress where the platform-specific `ix_cc_atmsar_async_vc_remove` function is called and then, the platform-specific function removes the ATM VCC.

To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointVccDelCBFunc`. In the callback function, if the status of the VCC delete operation is successful, then the ATM translator removes the VCC information from its VCC hash list. It also removes the information about this VCC from VCC list of the port to which this VCC is mapped.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context

- `response_data` Pointer to the packaged data from the backend API containing the ATM VCC ID information

Return Values

None

fpmIfATM_VccStatsGetFunc()

Syntax

```
Void fpmIfATM_VccStatsGetFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_get_vc_stats()`.

On a dual processor system like Angel Island, the CP-PDK runs on the ingress and the ATMSAR main CC resides on the egress processor. This function retrieves the VCC handle stored in ATM translators VCC hash list and then sends this VCC handle information to the egress where the platform-specific `ix_cc_atmsar_async_get_vc_stats` function is called andthen the platform-specific function gets the statistics of the ATM VCC.

To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointVccStatsGetCBFunc`.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to the packaged data from the backend API containing the ATM VCC ID information

Return Values

None

fpmIfATM_AAL2_ChannelSetFunc()

Syntax

```
Void fpmIfATM_IfAttrSetFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and checks if the VCC to which the AAL2 channel is mapped exists or not. In case the VCC exists, it invokes the platform-specific component `ix_cc_atmsar_async_cid_create()`.

On a dual processor system like the Angel Island, the CP-PDK runs on the ingress, and the ATMSAR main CC resides on the egress processor. This function internally sends the received AAL2 channel attribute information to the egress, where the platform-specific `ix_cc_atmsar_async_cid_create` function is called.

The platform-specific function in turn creates the AAL2 channel with the attributes passed to it. To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointAal2ChanSetCBFunc`. In the callback function, if the status of the AAL2 channel create operation is success, then the ATM translator adds the AAL2 information in the AAL2 hash list of the VCC to which the AAL2 channel is mapped. Each node in the AAL2 Hash list contains the following information:

- AAL2 channel Id
- AAL2 handle which is the handle returned by ATM SAR CC

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to packaged data from the backend API containing ATM AAL2 channel attributes information

Return Values

None

Note Currently, the `IXA_SDK_3.1 ATMSAR CC` supports only AAL5, and hence the CP-PDK support for AAL2 has not been integrated/tested with `IXA_SDK_3.1 ATMSAR` CC for AAL2.

fpmIfATM_AAL2_ChannelDelFunc()

Syntax

```
Void fpmIfATM_AAL2_ChannleDelFunc(FPPI_CBCORRELATOR
cbcorrelator,FPPI_CONTEXT context, void *response_data)
```

Description

This function extracts the information from the data pointed by the `response_data` pointer and invokes the platform-specific component `ix_cc_atmsar_async_cid_remove()`.

On a dual processor system like the Angel Island, the CP-PDK runs on the ingress whereas the ATMSAR main CC resides on the egress processor. This function retrieves the AAL2 channel handle stored in the ATM translators AAL2 hash list and then sends this AAL2 handle information to the egress, where the platform-specific `ix_cc_atmsar_async_cid_remove` function is called.

The platform-specific function removes the ATM AAL2 channel. To indicate the completion of the request, the platform-specific component calls the ATM translator callback function `LimePointAAL2ChanDelCBFunc`. In the callback function, if the status of the AAL2 channel delete operation is successful, then the ATM translator removes the AAL2 information from the AAL2 hash list of the VCC to which the AAL2 channel is mapped.

Parameters

- `cbcorrelator` Callback correlator that identifies the instance of the call
- `context` Callback context
- `response_data` Pointer to packaged data from the backend API containing AAL2 channel ID information

Return Values

None

Note Currently, the `IXA_SDK_3.1 ATMSAR` CC supports only AAL5, and hence the CP-PDK support for AAL2 has not been integrated/tested with `IXA_SDK_3.1 ATMSAR` CC for AAL2.

3.12 QOS Manager

This section gives an overview of the QOS FP plug-in manager.

3.12.1.1 Functionality

QOS manager provides mediation between the CP data and the SDK FP data.

This module maintains detailed information about the DPEs like handles, attributes, associations and DPE state. The DPE state takes any one of the following states:

- IDLE
- INUSE

During the creation of DPE, it is in the IDLE state and when it is associated with another DPE, its state becomes INUSE. The DPE can be deleted only in the IDLE state. All SDK APIs are used in the asynchronous mode.

1. On the request of `NPF_DS_DPE_CREATE`, new DPE state (`fpm_qos_dpe_t`) is created and stored in the DpeList. This state holds information about handles, state, number of associations, flow ID, port, and DPE attributes. On creation of the classifier, new flow ID is generated. No mechanism is provided to aggregate the different classifications into one flow ID. For each classifier element create, class id is generated based on DSCP value. If DSCP value is other than AF, BF or EF, `classid 15` is assigned. EF DSCP value is given high priority followed by AF class.
2. To add DSCP classifier, it is expected user must specify set of DSCP values in one request.
3. On callback from the SDK, response is generated to the CP. On error, the classifier entry is deleted from the DpeList. Refer `ix_cc_classifier_6t_api.h` and `ix_cc_classifier_6t_msg.h` for parameters details.
4. On the request of `NPF_DS_DPE_ADD_ASSOCIATE` between CLASSIFIER and METER, meter state is stored locally till the MARKER and METER association is made because SDK meter and the marker are embedded together. When the marker is associated with the meter, the SDK API is called to create meter entry. The METER DPE is created at the ingress or the egress depending upon on the interface direction field.
5. On the request of `NPF_DS_DPE_ADD_ASSOCIATE`, between the DROPPER and the CLASSIFIER, or DROPPER and MARKER, the `IX_CC_FPM` message is send to create the dropper entry at the egress side. Since the DROPPER is maintained at the egress side, all the DROPPER operations (`XXX_add_entry, XXX_update_entry, XXX_remove_entry, and XXX_statistics`) make use of the message communication between the ingress and the egress.

6. On the request of NPF_DS_DPE_DELETE_ASSOCIATE, between the DROPPER and the CLASSIFIER, or DROPPER and MARKER, the SDK API is called to remove the dropper entry and the state of DROPPER is set to IDLE.
7. On the request of the NPF_DS_DPE_DELETE_ASSOCIATE, between the MARKER and the METER, the state of the MARKER is set as IDLE and no SDK API is called. This is because of the embedding of the MARKER and the METER together.
8. On the request of the NPF_DS_DPE_DELETE_ASSOCIATE, between the METER and the CLASSIFER, the SDK API is called to remove the METER entry. On success, the state of the METER is set to IDLE.
9. When the DPE entry is created in the SDK, number of associations filed is incremented and the state is set to INUSE. When the DPE entry is deleted in the SDK, number of associations filed is decremented and the state is set to IDLE, when DPE has no associations.
10. On the request of NPF_DS_DPE_STATISTICS METER or MARKER DPE, the SDK METER request is made and the response is generated on the SDK callback.
11. On the request of NPF_DS_DPE_DELETE, entry is removed from the DpeList and SUCCESS is returned if the DPE state is IDLE. Otherwise, an error is returned.

3.12.1.2 Execution Context

All the QOS Manager down-calls execute in the context of the backend API callback functions registered during the initialization time. In the cases where the calls into the IXA SDK 3.51 are synchronous then the up-calls execute in the context of the backend API callback functions.

3.12.1.3 Initialization

The FP boot manager invokes the initialization function fpm_qos_mgr_init (). This function registers callbacks with the backend API for DPE operations.

3.12.1.4 Shutdown

The FP boot manager invokes the shutdown function qos_cm_mgr_shutdown (). This function de-registers the callbacks from the backend API and invokes the platform-specific shutdown function.

3.12.1.5 Data Structures

The following section describes the important data structures and description of the individual fields.

```
typedef struct fpm_qos_dpe
{
    QOS_DPE_State_t     state;
    uint32_t            assocs;
    NPF_DS_DpeHandle_t  dpeHandle;      /* DPE handle */
    NPF_DS_DpeHandle_t  prevDpeHandle;  /* DPE handle */
    NPF_DS_DpeDirection_t dir;          /* Direction,
INGRESS/EGRESS */
```

```
    ix_uint32           flowId;         /* Flow id */
    ix_uint32           classId;        /* Calss id, relative
Queue id */
    FPPI_PortID         port;           /* Port */
    NPF_DS_Dpe_t        dpe;            /* DPE attributes */
    void                *temp;          /* Used for temporary
storage */
} fpm_qos_dpe_t;
```

3.13 Run-time Configuration of the Forwarding Plane

The forwarding plane can be executed in the co-located mode with the control plane or in a remote mode on a different host. In case of the remote CP, the forwarding Plane executable is started from the command line with the following command:

```
ForwardingPlane <fe-name> <localhost-ip-address> <router_config>
```

The fe-name consists of the following two parts:

1. Name
2. An unique numeric ID

The numeric ID is the instance ID in the PDK namespace. So if the fe-name is `FE_0`, then the FE is located in namespace under /System/0/FE/0/

The `router_config` determines the behavior of the forwarding plane, whether it should behave as an IPv4 forwarder or as an ingress LER with QOS. The following `router_config(s)` are currently supported:

IPv4 Router: Configuration to do pure IPv4 routing. It requires a different Dispatch Loop.

In case of the co-located CP and FP, the CP must provide the above configuration settings options.