Protocol Support Service

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

intel

Contents

Protocol Support Service i

Contents iii

Part 1: Introduction 7

1 Introduction 9

1.1 Terminology 9

Table 1. Terminology table 9

1.2 References 10

Table 2. Reference table 10

Part 2: Control Plane PDK and Routing Protocols 11

2 Control Plane PDK and Routing Protocols 13

Figure 1: Control plane components of the CP-PDK 14

2.1 Virtual Interfaces and Choice of Tunneling Protocol 15

2.2 VIDD - Simulating Forwarding Plane Interfaces 16

Figure 2: VIDD virtualization effect 16

2.3 Forwarding Plane Support for Data Packet Transport 17

2.4 VIDD and the Packet Handler API 17

Part 3: Virtual Interface Controller (VIC) 19

3 Virtual Interface Controller (VIC) 21

3.1 Interactions with other PDK Modules 21

3.1.1 Execution context 21

3.1.2 Initialization 21

3.1.3 Shutdown 21

3.1.4 Managing Virtual Interfaces 22

Part 4: Virtual Interface Device Driver (VIDD) for Linux* 23

4 Virtual Interface Device Driver (VIDD) for Linux* 25

4.1 Design 25

4.2 Implementation Changes 26

4.2.1 Minor Changes 26

4.2.2 Major Changes 26

Figure 3: Data packet transmission 27

4.3 Creating and Manipulating Tunnels 28

4.3.1 Creating a New Tunnel 28

4.3.2 Setting a Tunnel's IP Address 29

4.3.3 Shutting Down and Deleting a Tunnel 29

4.3.4 Other Commands 29

Part 5: VIDD for VxWorks* 31

5 VIDD for VxWorks* 33

5.2 VIDD System Data Structures 33

5.2.1 DEV_OBJ – Device-Specific Control Object 33
5.2.2 END_ERR Data Structure 34
5.2.3 END_OBJ Data Structure 34
5.2.4 M2_INTERFACETBL Data Structure 35

5.3 VIDD Local Data Structures 36

5.3.1 pdk_vidd_physical_if_info 36
5.3.2 pdk_vidd_physical_if_node 36
5.3.3 pdk_vidd_ctrl 37

5.4 MUX Interface API 37

Table 3. System calls for MUX interface table 37

5.5 VIDD System Function Calls 38

5.5.1 pdk_vidd_npt_load() 39
5.5.2 pdk_vidd_npt_unload() 39
5.5.3 pdk _vidd_npt_start() 40
5.5.4 pdk_vidd_npt_stop() 40
5.5.5 pdk_vidd_npt_ioctl() 40

Table 4. IOCTL commands table 40

5.5.6 pdk_vidd_npt_send() 41
5.5.7 pdk_vidd_npt_mCastAddrAdd() 42
5.5.8 pdk _vidd_npt_mCastAddrDel() 42
5.5.9 pdk_vidd_npt_mCastAddrGet() 43
5.5.10 pdk_vidd_npt_pollSend() 43
5.5.11 pdk_vidd_npt_pollRcv() 44

Part 6: CE Packet Handler for VxWorks 45

6 CE Packet Handler for VxWorks 47

6.1 Execution Context 47
6.2 Initialization 47
6.3 Shutdown 48
6.4 On FE Bind 48
6.5 Packet Listen Thread 48
6.6 On FE Unbind 49
6.7 On Packet Receive from VIDD 49
6.8 Data Structures 49

Part 7: FE Packet Handler for VxWorks 51

7 FE Packet Handler for VxWorks 53

7.1 Design 53

7.1.1 Initialization 53
7.1.2 Incoming Packets 53
7.1.3 Outgoing Packets 54

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Udaya Shankar
2.1	Updated for Release 2.1	December 2003	Udaya Shankar
2.0	Updated for Release 2.0	August 2003	Udaya Shankar

Part 1: Introduction

1 Introduction

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

- Control plane
- Forwarding plan
- Management plane.

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

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

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

The introduction of standardized Application Program Interface (API) within the above-mentioned planes can help system vendors, Original Equipment Manufacturer (OEM), and end-users of these network elements to mix and match components available from different vendors to achieve a device of their choice. The Network Processing Forum (NPF) API is designed for this purpose, as it presents a flexible and well-known programming interface to the control plane applications. It makes the existence of multiple forwarding planes, as well as vendor-specific details, transparent to control plane applications.

The hardware properties and nature of interconnect used between the control and the forwarding planes are isolated. The protocol stacks and network processors available from different vendors can be easily integrated with the NPF APIs. The APIs included in the Control Plane Platform Development Kit (CP-PDK) are based on the NPF APIs. For more information about NPF, refer to http://www.npforum.org/.

This document describes the design of Protocol Support Services for the CP PDK. This includes the Virtual Interface Device Driver (VIDD), VIDD Controller (VIC) and control element (CE) and forwarding element (FE) packet handler modules of the CP PDK. The document is intended for developers of these modules and developers of other modules that make use of them.

1.1 Terminology

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

Table 1. Terminology table

Version	Date
ARP	Address Resolution Protocol
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 (www.omg.org)
CP-PDK	Control Plane Platform Development Kit
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
IXA	Internet eXchange Architecture
IXP, IXP 2000	Internet eXchange Processor, and a current instance of this processor. There are two versions of the IXP2X00 – IXP2400 with 8 microengines targeted at OC-48 POS line rates and IXP2800 with 16 microengines targeted at OC-192 POS line rates.
MPLS	Multiprotocol Label Switching
NPF	Network Processing Forum
OSPF	Open Shortest Path First (routing protocol)
RIP	Routing Information Protocol
Intel® XScale™ core	Forms the core of the IXP 2400 and 2800

1.2 References

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

Table 2. Reference table

Version	Date
[1]	Software Architecture Overview
[2]	Forwarding Plane Plugin API Reference
[3]	Route Cache Manager Design Reference
[4]	Configuration and Management Design Reference
[5]	Forwarding Plane Module Design Reference

Part 2: Control Plane PDK and Routing Protocols

2 Control Plane PDK and Routing Protocols

In order to support existing routing protocols and legacy control applications without modifications, the CP-PDK must provide additional support. This document describes the design of the approach of PDK to support legacy routing protocol stacks, such as, GateD, RIP, and OSPF, and executing in the control plane without any modifications. In the PDK, a set of modules, collectively termed Protocol Support Services, is responsible for providing this support.

This approach is based on the following:

Existing routing protocols and legacy applications use the socket interface to send and receive protocol/data packets. Internally, the socket interface uses the different physical interfaces available on the device, corresponding to physical ports, in a device-independent manner.

- **Virtual Interfaces:** If we can provide control plane applications with virtual interfaces that represent actual physical forwarding plane interfaces, routing protocols and other control plane applications will see the physical forwarding plane interfaces as virtual local interfaces on the control plane, allowing them to function without any modifications. This is accomplished in the CP-PDK by using a VIDD module that simulates all forwarding plane physical interfaces to the networking stack on the control plane.
- **Data packet transfer:** In addition to the virtualization of interfaces, packets directed to protocols/applications on the control plane need to be transported from the forwarding plane. Packets from the control plane that are transmitted to the virtual interfaces, are meant to be sent out to the corresponding physical interfaces on the forwarding plane. These packets are known as data packets. The exact mechanism and protocol that is used to transport or tunnel these packets is implementation dependent and is independent of virtual interfaces since virtualization of forwarding plane interfaces is essential for supporting unmodified control plane protocol stacks. In the CP PDK, such data packets are transported using an extension of the IP-in-IP protocol. The issues, advantages and disadvantages of this approach are detailed in section 2.1.
- **Synchronizing CE routing table updates with FE:** In addition to providing the above, a mechanism has to be provided to synchronize the routing table maintained in the control plane with the routing tables maintained on the individual forwarding elements. Routing stacks like GateD use the ioctl interface to populate the kernel FIB. With VIDD in place, routing protocols running on the control plane continue to add and delete routes in the kernel FIB of the control plane. While these routes are needed in control plane for correct functioning of the routing protocols, they are also required to be sent to the forwarding plane blades for configuration of the FIBs, so that packets can be forwarded. In order to distribute the route to forwarding plane FIBS, an application called route cache manager runs on top of CP-PDK and acquires these routes and invokes the IPv4 APIs. The route cache manager is described in more detail in [3]. The CP-PDK then handles the population of the FIBs on various forwarding plane blades with appropriate routes.

This process of virtualization and tunneling provides complete support for running unmodified legacy applications on the control plane. A high level architecture of the Linux* version of the PDK control plane components is shown in Figure 1.

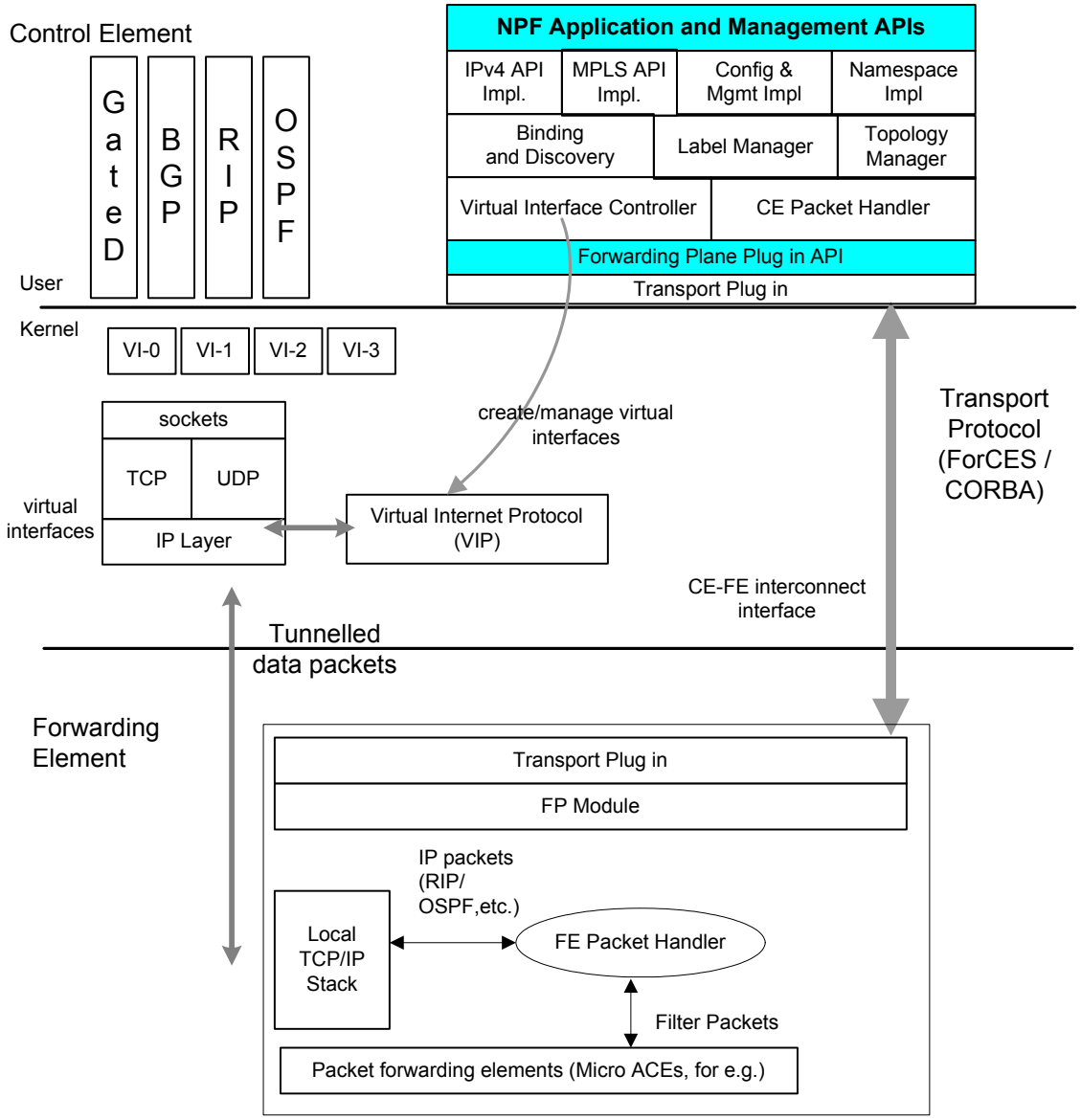

Figure 1: Control plane components of the CP-PDK

It is to be noted that simulating the FP interfaces on the CP is required only if the CP and FP are physically separated. If they are running in the same environment, virtual interfaces and are not required.

This document details the design of the following components:

- **Virtual Interface Controller (VIC)** – This is a module of the PDK that interacts with other components like namespace and configuration and management module to control the creation and management of the virtual interfaces.
- **Virtual Interface Device Driver (VIDD)** – This is a module that attaches virtual network interfaces to the network stack. In Linux*, this is part of a greater module that needs to be inserted into the kernel. In VxWorks*, it is built directly into the control plane.

- **Data Packet Handler** – There are two parts to the data packet handler, one for the control plane and one for the forwarding plane. The data packet handler transports all data packets between the two.

2.1 Virtual Interfaces and Choice of Tunneling Protocol

Virtual interfaces: Simulation of forwarding plane interfaces on the control plane is essential for maintaining the socket interfaces used by protocol stacks. The VIDD approach chosen by the PDK is generic and it provides a uniform abstraction to the higher layers of the networking stack of the control plane.

Note: The Linux* version of the VIDD needs to be a module in the kernel since VIDD has to create, delete and manage the virtual interfaces. For example, if a control plane management application changes the IP address of a forwarding plane interface, the VIDD module has to reflect the change in the corresponding virtual interface. In the PDK, VIDD is a kernel module controllable from user space.

Data packet transport: The mechanism and protocol used to transport control plane data packets, such as, routing PDUs, is implementation dependent. The CP-PDK uses a ForCES-based transport approach. Some of the options/issues are:

1. **Metadata:** Extra information needs to be maintained in most cases when transporting the data packet. For example, when transporting incoming multicast PDUs from the forwarding PLANE, the control plane has to know which forwarding plane interface the packet arrived on, so that higher-level protocols, such as, the multicast route daemon, can function correctly. This is the case for unicast IP packets also. Similar process takes place for multicast packets.For example, when OSPF sends IGMP join messages, it sends multicast packets out on specific interfaces. This information needs to be carried in the metadata for outgoing multicast packets.
2. **Transport mechanism:** The PDK uses an extension to IP-in-IP tunneling for tunneling. Data packets arriving on the forwarding plane are encapsulated in an external header that contains information about the forwarding plane interface, on which the packet has arrived. The packet is then sent across to the control plane using standard Internet Protocol messaging. A similar approach is used for packets originating from protocols on the control plane. The implementation is described later. Following is some more information other possible methods:

- **IP-in-IP tunneling:** This is a standard mechanism used for tunneling IP packets. An IP-in-IP tunnel simply adds an additional IP header that identifies the new destination of the packet (control plane if it is an incoming packet being forwarded by the forwarding plane) Corresponding ends of the tunnel compose/interpret the headers.

The channel for exchanging control packets between the control and forwarding planes could be used for transporting data packets also. Any proprietary protocol could be used to transport the packets to the control plane. For example, if an infiniband interconnect is being used, the data packets could be exchanged over the CE and FE endpoints of a virtual circuit.

- **Layer 2 tunneling:** An efficient Layer-2 tunneling protocol could also be used.
- ***Note:*** There are significant advantages/limitations of each approach. For example, in the current PDK implementation using IP-in-IP, data packets traverse the user-kernel boundary the least times – from user-space protocol stacks like GateD into the kernel, across the wire, into the kernel on the forwarding plane, back into user space into the FE Packet Handler,

and again into the kernel before heading out of the forwarding plane. This method however, requires a module to be added to the Linux kernel.

3. **Capturing protocol PDUs on the forwarding plane:** This is another factor that poses portability issues. Protocol PDUs like OSPF Hello/Join messages,. that arrive on the forwarding plane have to be captured before they can reach the kernel on the FP. If the packets reach the FP kernel, they will be dropped since the protocol they are intended for are actually executing on the CE. There are multiple ways of doing this:

- **Netfilter, IPChains:** This is a tool available in Unix*/Linux* environments. It allows for advanced packet capture facilities andit specifyies exactly what packets need to be captured, such as, destination IP address, destination port, and other header options. A limitation is that this has to be enabled and built into the kernel.
- **Raw sockets:** Raw socket allow the application to by pass the kernel TCP/IP stack whilesending or receiving data.
- **Packet Handler Core Component:** This is the approach of the CP PDK. Here, a special core component traps packets before they are sent to the kernel. The advantage of this approach is that it is guaranteed to work on any platform that IXA SDK is ported to.

2.2 VIDD - Simulating Forwarding Plane Interfaces

The control plane component of the CP-PDK consists of the NPF API implementations along with other components necessary for managing and abstracting multiple forwarding planes and different interconnects/transport mechanisms. It also contains the modules required in the PDK as well as the kernel for supporting unmodified protocol stacks as described earlier. A network element typically consists of a control plane blade and one of more forwarding plane blades. In Figure 2, each forwarding plane blade has five physical interface, such as, five 10/100 Ethernet ports. The routing protocols execute on the control plane blade and assume a socket interface. In order to preserve all the semantics of the socket interface, all the 10 physical interfaces have to be simulated to the IP stack on the control plane. All this simulation is handled by VIDD. As a result, the routing protocols in the control plane can be executed without any modifications to the PDU send and receive socket interface. Figure 2 shows what VIDD does pictorially. Subsequent sections describe in detail how this virtualization is achieved, the issues that have to be considered for this approach, the advantages and disadvantages of the approach.

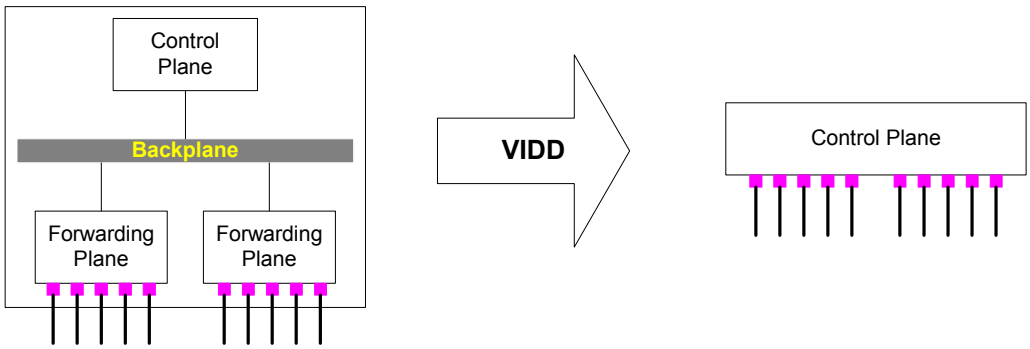

Figure 2: VIDD virtualization effect

2.3 Forwarding Plane Support for Data Packet Transport

On an IXA platform, all incoming packets are received by the ingress interface microblock and are then forwarded to other microblocks. Exception packets, that the micro-engines cannot handle, and control protocol PDUs need to be sent to the core processor for further processing. The rest of the packets are sent to the egress microblock, which transmits the packets over correct interface. In the core, control plane PDUs are received by a special core component called stack core component.

On the forwarding plane, the following support needs to be provided by the forwarding plane in order to handle data packets – this might be forwarding plane specific. The PDK uses the IXA platform for the forwarding plane. Irrespective of the actual transport mechanism, some additional support needs to be provided on both the control and forwarding planes to support data packets to and from legacy protocol stacks. Section 6 describes in detail the FE packet handling mechanism.

2.4 VIDD and the Packet Handler API

The packet handler API allows applications on the control plane to replace traditional socket send/receive calls with NPF packet handler API calls. When applications, such as, routing protocols are modified to use the packet handler API calls, VIDD need not be used in its entirety. When this happens, the virtual interfaces on the CE are not required any more since protocol stacks do not use the traditional socket interface anymore. However, the forwarding place still has to provide support to identify packets that need to be transported to the control plane. It also has to provide support for the transport itself, depending on the transport protocol to be used where applications receive them through the packet handler API.

Part 3: Virtual Interface Controller (VIC)

3 Virtual Interface Controller (VIC)

This module is responsible for controlling the dynamic creation and deletion of virtual interfaces on the CE. It interacts with other PDK components, such as, the C&M module and the namespace module. Interaction with namespace is not direct, but only for accessing data of FE, port, and interface objects. The following subsections describe the internal interaction with these modules and illustrate the implementation of this module using suitable pseudo code.

3.1 Interactions with other PDK Modules

The virtual interface controller registers the following callbacks with the C&M module. This API is internal to the PDK and is specified by the C&M module in [2]:

Callback for receiving notification interface events, such as, for creating, deleting, and managing virtual interfaces.

3.1.1 Execution context

All the functionality of this module executes in the context of the PDK. It does not create any threads or processes internally. The PDK manager performs initialization and shutdown. Subsequent execution of this module, for creating, managing or deleting virtual interfaces, is in the context of any events being generated by the PDK and will be executing in the corresponding context. For example, virtual interfaces are created/ deleted when C&M receives an event from the FE and invokes the corresponding callbacks registered.

3.1.2 Initialization

The PDK manager initializes the VIC module during the startup of the PDK. Since this module registers callbacks with the C&M module, it has to be initialized after C&M has been initialized.

```
vic_init
{
      register callback for NPF_EVENT_IF event
}
```

3.1.3 Shutdown

This module is shutdown by the PDK manager before C&M has been shutdown.

```
vic_shutdown {
      deregister callback for NPF_EVENT_IF event
      delete all virtual interfaces
}
```

3.1.4 Managing Virtual Interfaces

Virtual interfaces are created on the control plane for each port on an FE. When a FE binds to the CE, the CP-PDK initializes the FE data in the namespace and C&M modules. Subsequently, the C&M module downloads the IP addresses for the FE ports. When the FE reports a success for this operation, the virtual interface controller module creates the virtual interfaces on the CE, corresponding to each of the ports on the FE that the IP addresses were successfully assigned to. VIC creates virtual interfaces by making calls to VIDD, which may be IOCTL calls to the VIP kernel module in Linux*, or direct function calls to the VIDD module in VxWorks. This process is described in section 4.3.

The VIC module registers a callback with the C&M module that is to be invoked when an FE interface is configured with an IP address. This is the NPF_IF_IPADDR_CHANGE event.

Note: The same event callback can be used if the IP address of an interface changes at run-time, if changed by a configuration application. The pseudo code for this functionality is written below:

```
vic_onInterface_Event (NPF_EVENT event, NPF_HANDLE intfhandle ) {
switch (event)
case NPF_EVT_IF_UP:
if (there is no virtual interface yet for this interface) {
generate a unique name – vi0, vi1, etc.
construct the virtual interfaces parameters (vif_params)
tell VIDD to construct new interface
configure IP address
}
else {
set the device to UP and RUNNING
}
break;
case NPF_EVT_IF_DOWN:
set the device state to DOWN
break;
case NPF_EVT_IPADDR_CHANGE:
update device IP address
break;
case NPF_EVT_IF_DELETE:
open the VIDD
make ioctl to delete the virtual interface
break;

}
```

*Part 4: Virtual Interface Device Driver (VIDD) for Linux**

4 Virtual Interface Device Driver (VIDD) for Linux*

The main requirements for the VIDD module are:

- It should simulate the physical interfaces on the different forwarding planes at the CE.
- Depending on the transport mechanism used, the VIDD module has to provide the ability to transmit and receive data packets to/from the forwarding plane. For example, if using IP-in-IP tunneling, when any application on the CE transmits packets to be sent out any of these virtual interfaces, the VIDD module is responsible for tunneling those packets, using an appropriate tunneling protocol to the correct forwarding plane where the physical interface is. When the CE receives tunneled packets from an FE, VIDD module should intercept them, de-tunnel the packet, that is, interpret the tunnel header appropriately, and present the packet to the networking stack for normal delivery to the correct application.
- Meta data required by the forwarding plane must be included with every packet.

The rest of this chapter describes the VIDD design for both the CE and the FE. Since there is no need to simulate the physical interfaces at the FE, the section on creating new tunnels is valid only for the CE.

4.1 Design

The Linux* kernel already has the ability to create and use virtual network interfaces through its own IP-in-IP tunneling support. This tunneling supports almost all of the requirements of the PDK VIDD, except for the metadata that needs to be sent with every packet. Therefore, we have chosen to take the Linux* implementation of IP-in-IP and extend it to include our metadata.

The IP-in-IP implementation in Linux* that we chose to extend can be found in the kernel source in net/ipv4/ipip.c.

The basic idea behind the tunneling support in Linux is:

- To register a new network device
- When receiving packets from the network stack for transmission, add an new IP header with source and destination set to the ends of the tunnel and return it to the network stack
- When receiving packets from the network stack for local delivery, remove the extra IP header and give packet back to the network stack

Since the majority of all the functionality required by the PDK is already present in the IP-in-IP module in the Linux* kernel, the rest of this section will discuss only the changes made, and how to use the VIP module.

Since the IP-in-IP module already performs all the requirements of the packet handler, there is no separate packet handler required on the Linux* control plane.

4.2 Implementation Changes

The following sections describe the required implementation changes to the IP-in-IP module included in the Linux* kernel source distribution to satisfy the requirements of PDK.

4.2.1 Minor Changes

A copy of `ipip.c` must be placed into the PDK code base. Since it does the entire packet handling, it should be put into the PacketHandler sub-directory. Since the PDK's current build system is geared towards building a single application, a new stand alone makefile will be required to make it easy to build a kernel module.

In order to avoid name clashes, or confusion on the part of the programmer, all global functions and variables must be renamed. The original `ipip.c` names are all globals with a prefix of `ipip`, therefore all globals in the PDK's `vip.c` version will be prefixed with `vipip`.

IP-in-IP defaults to a network name of `tunl0`, in order to avoid confusion and namespace clashes and to maintain the same convention as earlier versions of the PDK, VIP needs to change this to `vi0`.

4.2.2 Major Changes

All IP-in-IP packets are transported using protocol 4, any IP packets of that protocol coming into a system are automatically given to the IP-in-IP module. Linux* has an IP-in-IP protocol defined as `IPPROTO_IPIP in /usr/include/netinet/in.h`. Since we need all incoming VIP packets to come to our module, instead, we need to register a new protocol with the system. This protocol is `IPROTOP-VIP`. All packets are sent with this protocol. Once the VIP module is loaded, any packets with a protocol of 105 will be directed to the VIP module. Since we have not registered this protocol number and application of it with any governing body, there is a possibility of getting packets from some other network stack that also uses 105. This has a fairly low probability, as both endpoints in the system are controlled.

All packets given to the IP-in-IP module for transmission to the network are encapsulated in a new IP header. This is enough to enable tunneling, but not enough for the PDK's purposes, as detailed above. Therefore, an additional header will need to be added. This header will contain the id of the port that the packet must leave the FE, and, for redundancy, the length of the tunneled packet. Here is a pseudo code declaration of the header:

```
vip_header {
uint8 portid;
uint8 length;
}
```

Conversely, all VIP tunneled packets entering the system must have the IP header and VIP header stripped off of them before reentering the network stack. The following diagram shows a data packet before and after encapsulation and which parts would be contributed by the existing IP-in-IP code and which part needs to be added by VIP.

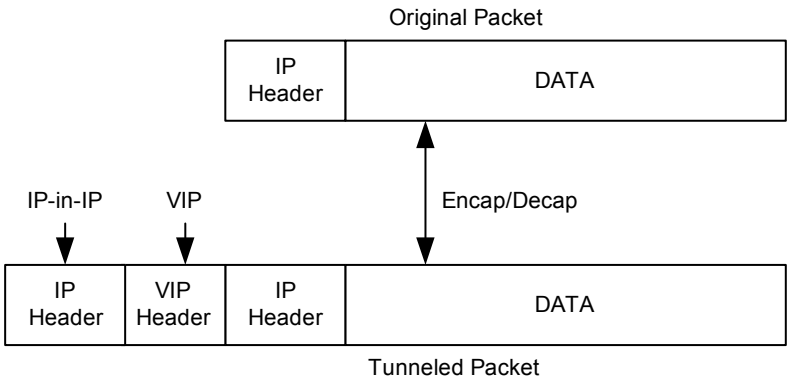

Figure 3: Data packet transmission

Currently, the IP-in-IP module creates new tunnels when it receives an IOCTL of type SIOCADDTUNNEL. This IOCTL is accompanied by a data structure of type ip_tunnel_parm. This structure contains all the information IP-in-IP needs to create tunnel. Since VIP requires `portid` as well, a new structure will have to be created that extends `ip_tunnel_parm` to include `portid`.

```
struct vip_tunnel_parm
{
/* original ip_tunnel_parm fields */
char        name[IFNAMSIZ];
int         link;
__u16       i_flags;
__u16       o_flags;
__u32       i_key;
__u32       o_key;
struct iphdr      iph;
/* vip addition */
int         portid;
}
```

The IP-in-IP implementation saves the parameter with its internal tunnel lists. VIP will only have to use the `portid` at the appropriate time later.

4.3 Creating and Manipulating Tunnels

The PDK already has a module for creating and controlling virtual interfaces, the `vidd_controller.c`, this file is updated to work with the new VIP module.

Since VIP is basically a network device from the Linux* kernel's perspective, it needs to be acessed through the same methods.

All commands to the VIP module must be sent via IOCTLs through a network socket. The network socket handle is created this way:

```
sockfd = socket(AF_INET,SOCK_DGRAM,0);
```

All IOCTLs are sent through the socket with the same parameter type, with different values and data set to it. All IOCTL calls are of the form:

```
struct ifreq ifr;
ifr.ifr_ifru.ifru_data = (void*)&someDataToSend;
ioctl(skfd,SIOCTLCOMMAND,&ifr);
```

The following subsections outline how to perform various operations on VIP.

4.3.1 Creating a New Tunnel

Creating a new VIP tunnel will be accomplished by sending the `SIOCADDTUNNEL` command to the VIP module. A name for the tunnel, its destination and source addresses, the portid, and other information will need to be provided via the `vip_tunnel_parm` structure. Here is some sample code describing this process:

```
int skfd;
struct ifreq ifr;
struct vip_tunnel_parm vtp

vtp.iph.version = 4;
vtp.iph.ihl = 5;
vtp.iph.frag_off = htons(0x4000);   /* don't fragment */
vtp.iph.protocol = IPPROTO_VIP;     /* the new vip protocol, 105 */
vtp.iph.saddr = INADDR_ANY;
vtp.iph.daddr = ipaddrOfForwardingPlane;
vtp.portid = idOfPortOnFP;
strncpy(vtp.name,nameOfVirtualDevice,IFNAMSIZ-1);

strcpy(ifr.ifr_name,"vi0");   /* vi0 is the name of the VIP module */
ifr.ifr_ifru.ifru_data = (void*)&vtp;   /* all commands are sent via struct
ifreq ifr */

sockfd = socket(AF_INET,SOCK_DGRAM,0);
ioctl(skfd, SIOCADDTUNNEL,&ifr);
close(skfd);
```

4.3.2 Setting a Tunnel's IP Address

Once a tunnel is created, it has the remote and local IP addresses set that comprise the tunnel that the packets flow through, but the address of the virtual device is not set. The controller will need to give it the same address as the remote adapter on the forwarding plane that it is representing. This is done through an IOCTL through a network socket. Following is the sample code for this:

```
int skfd;
struct ifreq ifr;
struct sockaddr_in sin;

strncpy(ifr.ifr_name,nameGivenToVirtualDevice,IFNAMSIZ-1);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = ipaddrOfDeviceOnFE;
memcpy(&(ifr.ifr_addr),&(sin),sizeof(struct sockaddr));

skfd = socket(AF_INET,SOCK_DGRAM,0);
ioctl(skfd,SIOCSIFADDR,&ifr);
close(skfd);
```

4.3.3 Shutting Down and Deleting a Tunnel

As ports go down on the Forwarding Plane, the Control Plane needs to remove the virtual devices that represent them. This will be done along the lines of the following sample code:

```
int skfd;
struct ifreq ifr;
struct vip_tunnel_parm vtp;
strncpy(vtp.name,nameOfTunnelToDelete,IFNAMSIZ-1);
strcpy(ifr.ifr_name,"vi0");
ifr.ifr_ifru.ifru_data = (void*)&vtp;
skfd = socket(AF_INET,SOCK_DGRAM,0);
ioctl(skfd,SIOCDELTUNNEL,&ifr);
close(skfd);
```

4.3.4 Other Commands

All other commands that need to be sent to VIP or performed on the virtual devices, such as, setting `netmask`, will be done in the same way.

*Part 5: VIDD for VxWorks**

5 VIDD for VxWorks*

The VIDD for VxWorks has the same design requirements as its counterpart for Linux. The goal is to expose the FE ports as regular network interfaces to the OS TCP/IP stack. The VIDD is a network device driver. There are two types of network drivers under VxWorks:

- Enhanced network drivers
- Traditional network drivers

5.1.1.1 Enhanced Network Drivers

An Enhanced Network Driver (END) is a data-link-layer driver model that uses MUX functions to communicate with network protocols. By registering with the MUX interface, END drivers have greater flexibility to work with different network protocols. The MUX is an OS layer through which network protocols communicate with the data-link layer. Traditional model drivers communicate directly with protocol layer.

The MUX-based model for network drivers contains standardized entry points that are not present in the traditional drivers. VIDD on VxWorks is an END, and conforms to MUX APIs. The IXA SDK VIDD will be implemented as an Network Protocol Toolkit (NPT) driver, which has the functionality of an END except that it deals with IP packets rather than Ethernet packets.

5.1.1.2 Traditional Network Drivers – drivers supporting 4.3 BSD driver interface

This driver supports the 4.3 BSD driver interface and is tightly coupled with upper-level network protocols. It does not use the MUX interface, thus, portability across network protocols could be an issue.

Other types of device drivers are out of the scope of this design.

The VxWorks* OS has a flat memory model without division between kernel and user mode. This leads to availability of threads and pre-emptive multitasking for VxWorks* device drivers. This is different from the Linux* driver model, which is not preemptive, runs to completion, and does not have threads, and does not lock the data from synchronous access.

5.2 VIDD System Data Structures

These data structures are required by the VxWorks* kernel to be implemented by any network driver.

5.2.1 DEV_OBJ – Device-Specific Control Object

The `DEV_OBJ` structure is the glue linking the device generic `END_OBJ` structure, defined in section 5.2.3, with the device specific data object referenced by pDevice. The VxWorks* kernel defines this.

```
typedef struct dev_obj
{
char name[END_NAME_MAX]; /* device name */
int unit;    /* unit number of the interface */
char description[END_DESC_MAX]; /*driver description */
void* pDevice; /* Pointer back to the device data. */
} DEV_OBJ;
```

5.2.2 END_ERR Data Structure

This is the error data structure that holds system and user-defined errors and is used to pass errors from VIDD to the CP protocol.

```
typedef struct end_err
{
IX_UINT32   errCode; /* Error code */
      char* pMesg; /* NULL –terminated error message */
      void* pSpare /*user defined data */
} END_ERR;
```

The errCode member of the END_ERR structure is 32 bits long. The lower 16 bits are reserved for system error messages, while the upper 16 bits may be used for custom error messages. The currently defined error codes are:

END_ERR_INFO	This error is informational only.
END_ERR_WARN	A non-fatal error has occurred.
END_ERR_RESET	An error occurred that forced the device to reset itself but the device has recovered.
END_ERR_DOWN	A fatal error occurred that forced the device to go down. The device can no longer send or receive packets.
END_ERR_UP	The device was down but has now come up and may again send and receive packets.

5.2.3 END_OBJ Data Structure

The core data structure for a device is the END object, or END_OBJ. The driver allocates this structure and initializes some of its elements within the `endLoad()` function. END_OBJ is the basic END object from which everyone derives. This data structure defines a device-independent amount of state that is maintained by all drivers/devices. Each specific device derives from this object first and then incorporates its own data structures.

```
typedef struct end_object
{
```

NODE node; /* The root of the device hierarchy. The MUX sets the value of this member */

```
    DEV_OBJ devObject;
    STATUS (*receiveRtn) ();    /* Routine to call on reception. */
  struct net_protocol *outputFilter;/* Optional output filter routine. */
    void* pOutputFilterSpare;          /* Output filter's spare pointer */
    BOOL attached;                     /* Indicates unit is attached. */
    SEM_ID txSem;                /* Transmitter semaphore. */
    long flags;                        /* Various flags. */
    struct net_funcs *pFuncTable;      /* Function table. */
    M2_INTERFACETBL mib2Tbl;    /* MIBII counters. */
    LIST multiList;                    /* Head of the multicast address list */
    int     nMulti;                    /* Number of elements in the list. */
    LIST protocols;                    /* Protocol node list. */
    int snarfCount;       /* Number of snarf protocols at head of list */
    NET_POOL_ID pNetPool;       /* Memory cookie used by MUX buffering. */
    M2_ID * pMib2Tbl;                  /* RFC 2233 MIB objects */
} END_OBJ;
```

5.2.4 M2_INTERFACETBL Data Structure

An M2_INTERFACETBL structure tracks the MIB-II variables used in the driver. The driver must initialize this structure, although the elements in the structure may be used and adjusted both by the driver and by the MUX.

```
typedef struct M2_INTERFACETBL_TAG
{
int ifIndex;
char ifDescr[M2DISPLAYSTRSIZE];  /* a text description. */
long ifType;  /* type of device, from 1158. */
long ifMtu;  /* maximum packet size. */
unsigned long ifSpeed;  /* speed in bits/sec */
M2_PHYADDR ifPhysAddress;  /* LLC address. */
long ifAdminStatus;  /* UP/DOWN/TEST */
long ifOperStatus;  /* UP/DOWN/TEST */
unsigned long ifLastChange;  /* last change of the iface */
unsigned long ifInOctets;  /* # of received octets */
unsigned long ifInUcastPkts;  /* # of unicast packets received */
unsigned long ifInNUcastPkts;  /* # of broad/multicast pkts recd */
unsigned long ifInDiscards;  /* # of input discards */
unsigned long ifInErrors;  /* # of input errors */
unsigned long ifInUnknownProtos;  /* # of unknown packets */
unsigned long ifOutOctets;  /* # of octets sent */
unsigned long ifOutUcastPkts;  /* # of unicast packets sent */
unsigned long ifOutNUcastPkts;  /* # of broad/multicast pkts sent */
```

```
unsigned long ifOutDiscards; /* # of packets discarded */
unsigned long ifOutErrors; /* # of output errors */
unsigned long ifOutQLen; /* size of the output queue */
M2_OBJECTID ifSpecific; /* defs specific to media used */
} M2_INTERFACETBL;
```

5.3 VIDD Local Data Structures

This section provides information on the data structures used by the VIDD module.

5.3.1 pdk_vidd_physical_if_info

```
typedef struct pdk_s_physical_if_info
{
    uint32_t feId;
    uint32_t portId;
    uint8_t name[IFNAME_LEN];
    uint32_t unit;
    uint8_t macAddr[ETH_ALEN];
    uint32_t ipv4Addr;
    uint32_t ipv4Mask;
    uint8_t if_status;
    media_type_t mediaType;
    uint16_t MTU;
    uint32_t linkSpeed;
} pdk_physical_if_info;
```

5.3.2 pdk_vidd_physical_if_node

This list of data structures is created upon initialization by the VIDD and represents the hardware network interfaces on the FEs.

```
typedef struct pdk_s_vidd_physical_if_node
{
    END_OBJ end; /* corresponding END_OBJ struct */
    void* pMuxCookie; /* pointer to the cookie used for MUX communication */
    pdk_vidd_ctrl* pViddCtrl; /* circular reference to VIDD control structure */
    pdk_physical_if_info* pPhysicalIfInfo; /* Pointer to port info */

  /* next interface structure in the list of the structures. */
    pdk_vidd_physical_if_node* pNextPhysicalIf;
} pdk_vidd_physical_if_node;
```

5.3.3 pdk_vidd_ctrl

This structure is defined by the driver and is created during initialization. It holds the control information about memory pool and is used by the Packet Handler module.

```
typedef struct pdk_s_vidd_ctrl
{
    /**
     * Memory cookie used by MUX buffering - this identifies
     * the buffer pool that is shared among all VIDD interfaces.
     */
    NET_POOL_ID pNetPool;
    /**
     * Pointer to Mblk memory area used to create the buffer pool.
     * We must store this pointer at initialization and free its
     * memory at shutdown.
     */
    void* pMclBlkCfg;
    /**
     * Pointer to cluster memory area used to create the buffer
pool.
     * We must store this pointer at initialization and free its
     * memory at shutdown.
     */
    void* pClustMem;
} pdk_vidd_ctrl;
```

5.4 MUX Interface API

The VIDD driver must implement the following system calls to support the MUX interface:

Table 3. System calls for MUX interface table

System Call	Description
`nptLoad()`	Load a device into the MUX and associate a driver with the device.
`nptUnload()`	Release a device, or a port on a device, from the MUX.
`nptSend()`	Accept data from the MUX and send it on towards the physical layer.
`nptMCastAddrAdd()`	Add a multicast address to the list of those registered for the device.
`nptMCastAddrDel()`	Remove a registered multicast address from the list of those registered for the device.
`nptMCastAddrGet()`	Retrieve a list of multicast addresses registered for a device.
`NptPollSend()`	Send frames in polled mode rather than interrupt-driven mode.

System Call	Description
nptPollReceive()	Receive frames in polled mode rather than interrupt-driven mode.
nptStart()	Connect device interrupts and activate the interface.
nptStop()	Stop or deactivate a network device or interface.
nptIoctl()	Support various ioctl commands.

The functions are defined inside the endLoad() function by allocating the NET_FUNCS data structure. The MUX uses this structure to reference the functions implemented for a driver. The NET_FUNCS structure is defined as follows:

```
typedef struct net_funcs
{
STATUS (* start)( END_OBJ*);
STATUS (* stop)( END_OBJ*);
STATUS (* unload)( END_OBJ*);
int (* ioctl)( END_OBJ*, int, caddr_t);
STATUS (* send)( END_OBJ*, M_BLK_ID);
STATUS (* mCastAddrAdd) (END_OBJ*, char*);
STATUS (* mCastAddrDel) (END_OBJ*, char*);
STATUS (* mCastAddrGet) (END_OBJ*, MULTI_TABLE*);
STATUS (* pollSend)( END_OBJ*, M_BLK_ID);
STATUS (* pollRcv) (END_OBJ* pEND, M_BLK_ID pMblk,
long* pNetSvc, long* pNetOffset,
void* pSpareData)
M_BLK_ID (* formAddress) (M_BLK_ID, M_BLK_ID, M_BLK_ID);
STATUS (* packetDataGet) (M_BLK_ID, LL_HDR_INFO *);
STATUS (* addrGet)(M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID,
M_BLK_ID);
} NET_FUNCS;
```

The fields in this structure correspond with entry points for system calls for the VIDD driver.

5.5 VIDD System Function Calls

These function calls are implemented by the VIDD in order to conform to the MUX/END interface. These functions control all interactions between the VIDD and VxWorks*:

- configuration and initialization
- shutdown processing
- packet receiving
- packet sending
- memory allocation
- ioctl handling

The addresses of functions are in the END_OBJ structure so they can be called from the MUX library.

5.5.1 pdk_vidd_npt_load()

This function creates a logical interface on the VIDD side and registers it with the VxWorks* MUX layer. For initial release one logical interface for each physical interface is supported. Future releases may support multiple logical interfaces per port. The user does not call this function directly – the VxWorks* MUX layer uses it as an entry point when adding a new physical interface. This function is implemented as a two-pass algorithm – the MUX layer calls this function twice – once with an empty initialization string, and then with the real initialization string.

Syntax

```
END_OBJ* pdk_vidd_npt_load(char* initString, void* pBsp);
```

Input Parameters

`void* pBsp` Optional BSP-specific information. This is used as a context, in this case a pointer to the VIDD control structure.

Input/Output Parameters

`char* initString` On the first pass of this function, initString is passed in as an empty allocated string, and the base name of the interface (e.g. “eth”) is copied into it. On the second pass, initString contains the interface parameters for the logical interface being loaded. The function will parse initString to fill in the logical interface data structure.

Return Values

Pointer to the new END_OBJ data structure.

NULL return value indicates an error.

5.5.2 pdk_vidd_npt_unload()

This function is called by the MUX to “release” the device. The function is called for each port that has been activated by call to the pdk_vidd_npt_load(). This function will only free the memory allocated to the device data members – another function will handle the housekeeping tasks of unlinking this device from the list of interfaces and decrementing the overall interface count. That is, the scope of this function is limited to the device itself – the larger implications of unloading a device will be handled elsewhere. In this release this function does not have to do anything.

Syntax

```
STATUS pdk_vidd_npt_unload(END_OBJ* pEnd);
```

Input Parameters

`END_OBJ* pEnd` Pointer to END_OBJ data structure allocated by the Load() function.

5.5.3 pdk _vidd_npt_start()

The Start() function brings up the network interface specified by the END_OBJ structure and makes the interface active and available to the OS.

Syntax

```
STATUS pdk_vidd_npt_start (END_OBJ* pEnd);
```

Input Parameter

`END_OBJ* pEnd` Pointer to END_OBJ data structure allocated by `Load()` function.

Return Values

OK

ERROR

5.5.4 pdk_vidd_npt_stop()

This function brings down the interface specified by the END_OBJ structure and deactivates the interface.

Syntax

```
STATUS pdk_vidd_npt_stop (END_OBJ* pEnd);
```

Inputs

Pointer to END_OBJ data structure allocated by Load() function.

Return Values

OK

ERROR

5.5.5 pdk_vidd_npt_ioctl()

The VIDD needs to support IOCTL commands to keep the IOCTL interface with existing network protocols. The following table gives the list of commonly used IOCTL commands.

Table 4. IOCTL commands table

Command	Function	Data Type	Supported
SIOCGIFMTU	Get MTU	char*	Yes
EIOCSFLAGS	Set device flags	int	Yes
EIOCGFLAGS	Get device flags	int	Yes

Command	Function	Data Type	Supported
EIOCSADDR	Set device address	char*	No
EIOCGADDR	Get device address	char*	Yes
EIOCMULTIADD	Add multicast address	char*	No
EIOCMULTIDEL	Delete multicast address	char*	No
EIOCMULTIGET	Get multicast list	MULTI_TABLE*	No
EIOCPOLLSTART	Set device into polling mode	NULL	No
EIOCPOLLSTOP	Set device into interrupt mode	NULL	No
EIOCGFBUF	Get minimum first buffer for chaining	int	Yes
EIOCGHDRLEN	Get the size of the data link header	int	Yes
EIOCGNPT	Query a driver to determine whether it is a NPT driver	int	Yes
EIOCGMIB2233	Retrieves the RFC2233 MIB II table	M2_ID *	Yes
EIOCGMIB2	Get the MIB-II counters from the driver	char*	Yes

In addition to the standard commands, which are in the range 0-128, the application or network protocol can define its own IOCTL commands to communicate with the driver.

Syntax

```
int pdk_vidd_npt_ioctl(END_OBJ* pEnd, int command, caddr_t
buffer);
```

Input Parameters

`END_OBJ* pEnd` pointer to the END_OBJ structure of the interface

`int command` ioctl command.

Input/Output Parameters

`caddr_t buffer` character buffer holding response from the command.

Return Values

OK

EINVAL – command is not supported.

5.5.6 pdk_vidd_npt_send()

The MUX interface calls this function when the network protocol is sending a network packet. Network buffers are represented by an mBlk chain in VxWorks. This function takes the packet

stored in the mBlk and performs all necessary checks on the packet before calling the packet handler API ph_write_packet() function.

Syntax

```
STATUS pdk_vidd_npt_send(END_OBJ* pEnd, M_BLK_ID pMblk, char*
dstMacAddr, long netType, void* pSpare);
```

Input Parameters

`END_OBJ* pEnd` pointer to the END_OBJ structure. It identifies the transmit interface.

`M_BLK_ID pMblk` mBlk chain containing the network buffer. The data represents the full link layer frame.

`char* dstMacAddr` destination MAC address from the OS stack. Ignored by this function, as the packet handler will take care of this.

`long netType` network service type.

`void* pSpare` optional network service data.

Return Values

OK

ERROR

5.5.7 pdk_vidd_npt_mCastAddrAdd()

This function adds a new link-layer multicast address to the table of multicast addresses for the interface – in the initial release this call does not affect the underlying hardware.

Syntax

```
STATUS pdk_vidd_npt_mCastAddrAdd (END_OBJ * pEnd, char* pAddress);
```

Input Parameters

`END_OBJ* pEnd` pointer to the END_OBJ structure to help identify the port.

`char* pAddress` physical address to add to the Multicast table.

Return Values

OK

ERROR

5.5.8 pdk _vidd_npt_mCastAddrDel()

The function removes previously added link-layer multicast address – in the initial release this does not affect the underlying hardware.

Syntax

```
STATUS pdk_vidd_npt_mCastAddrDel (END_OBJ* pEnd, char* pAddress);
```

Input Parameters

`END_OBJ* pEnd`	pointer to the END_OBJ structure to help identify the port.
`char* pAddress`	physical address to delete from the Multicast table.

Return Values

OK

ERROR

5.5.9 pdk_vidd_npt_mCastAddrGet()

This function gets the list of all multicast addresses that are active on the interface.

Syntax

```
STATUS pdk_vidd_npt_mCastAddrGet (END_OBJ * pEnd, MULTI_TABLE*
pTable);
```

Input Parameters

`END_OBJ* pEnd`	pointer to the END_OBJ structure to help identify the interface.
`MULTI_TABLE* pTable`	pointer to the structure where the list will be put.

Return Values

OK

ERROR

5.5.10 pdk_vidd_npt_pollSend()

Polling-mode equivalent to the send() routine – this is not supported in the current implementation and always returns ERROR.

Syntax

```
STATUS pdk_vidd_npt_pollSend (END_OBJ* pEND, M_BLK_ID pPkt, char*
dstAddr, long netType, void * pSpareData);
```

Input Parameters

`END_OBJ* pEnd`	pointer to the END_OBJ structure. It identifies the transmit interface.
`M_BLK_ID pMblk`	mBlk chain containing the network buffer. The data represents the full link layer frame.
`char* dstMacAddr`	destination MAC address from the OS stack. Ignored by this function, as the VIDD will do its own route table lookup to obtain the destination MAC address.
`long netType`	network service type.
`void* pSpare`	optional network service data.

Return Values

ERROR

5.5.11 pdk_vidd_npt_pollRcv()

The current implementation of VIDD does not support PollReceive(). Therefore this function always returns an error.

Syntax

```
STATUS pdk_vidd_npt_pollRcv (END_OBJ* pEND, M_BLK_ID pMblk, long*
pNetSvc, long* pNetOffset, void* pSpareData);
```

Input Parameters

`END_OBJ* pEnd`	pointer to the END_OBJ structure. It identifies the transmit interface.
`M_BLK_ID pMblk`	mBlk chain containing the network buffer. The data represents the full link-layer frame.
`long* pNetSvc`	payload/network frame type.
`long* pNetOffset`	offset to network frame.
`void* pSpare`	optional network service data.

Return Values

ERROR

Part 6: CE Packet Handler for VxWorks

6 CE Packet Handler for VxWorks

The CE packet handler is a module of the PDK and it interacts with VIDD module. It is responsible for:

- Injecting protocol data packets tunneled from the FE into the corresponding virtual interface.
- Receiving data packets being sent on any/all of the virtual interfaces and tunneling it to the FE for subsequent transmission.

In the PDK, a simple IP-in-IP protocol is used to transport the data packets between the control and forwarding planes. This module uses the existing IP network stack to transmit and receive packets thereby avoiding any extraneous code requirements in the PDK.

The CE packet handler registers a callback with the transport plug-in module in order to receive callbacks when:

- An FE binds, to start listening for packets from it, and to store information needed to send packets to it.
- An FE unbinds, to remove it from consideration.

6.1 Execution Context

This module executes in two contexts:

- **PDK context:** This module is initialized and shutdown by the PDK manager. The callbacks it registers with VIDD controller and FP plug-in execute in the corresponding contexts.
- **Packet Listen Thread:** This thread is created on initialization of the module. The main responsibility of this thread is to listen for packets being transmitted from the FE to the CE destined for the virtual interfaces created on the CE. As described in the previous section, each virtual interface exposes a character device API, thus allowing a user space application to perform read operations on it. The exact manner in which this is done will be described subsequently. This thread is shutdown during the shutdown of the PDK.

6.2 Initialization

The PDK manager initializes this module during PDK startup. Since this module registers callbacks with the VIDD Controller module, it has to be initialized after the VIDD controller has been initialized.

```
ph_init {
```

register callback with TP for `FE_BIND` event

register callback with TP for `FE_UNBIND` event

create a new socket of protocol 105 (VIP)

start `Pkt_Listen` thread

}

6.3 Shutdown

The PDK manager shuts down this module before the VIDD controller is shutdown.

`ph_shutdown {`

deregister callback with TP for `FE_BIND` event

deregister callback with VIC TP for `FE_UNBIND` event

stop `Pkt_Listen` thread

close socket

}

6.4 On FE Bind

The packet handler module registers this callback with the TP module. When TP accepts a bind from a new FE, it informs the packet handler about this.

The callback updates a local data structure and adds the newly connect FE to the list of FE for which the packet listen thread has to listen. The packet listen thread waits for packets transmitted from any of the FE. To reduce time requirements for looking up FE information, this list is maintained as a hash table.

6.5 Packet Listen Thread

The main responsibility of this thread is to listen for packets transmitted to the CE from all the FEs on protocol 105, known as VIP. As each packet is read from the network, the source FE is determined and the thread passes the encapsulated packet on to the VIDD passing in the FEID and `Portid`.

```
ph_thread () {
      while  (1) { /* loop forever until stopped */
      receive packet from socket
      if (received packet ) {
lookup address in FE list
remove ip_header from packet
remove vip_header from packet
retrieve portid from vip_header
```

```
pass packet to VIDD with FEID and portid
      }
}
```

6.6 On FE Unbind

The packet handler module registers this callback with the TP module. When TP unbinds from an FE, it informs the packet handler about this.

The callback removes the FE from the list of interfaces FEs the packet listen thread has to listen for. The packet listen thread will subsequently start ignoring packets from this FE. This interaction is illustrated in the pseudo code that follows.

```
ph_on_newFE() {
      acquire lock to update list of FEs
update list of FEs – delete FE
      release lock
}
```

6.7 On Packet Receive from VIDD

The packet handler registers this callback with the VIDD module for getting notified when a data packet arrives from any of the virtual devices. The VIDD additionally provides metadata information, such as, the FE the packet is intended for and the physical port it go out on the FE.

Packets received through the VIDD API will have to be converted from MBLKs to character buffers, encapsulated in a VIP header, and sent to the network in the normal manner. This interaction is illustrated in the following pseudo code.

```
ph_write_packet( FEID fe, port_id port, M_BLK_ID pMblk ) {
lookup FE information in FE list
convert pMblk to char buffer using netMblkToBufCopy

add a vip_header to the char buffer
set portid and length appropriately
send the packet to the address of the FE
}
```

6.8 Data Structures

This structure is maintained by the packet handler and is shared by the packet listen thread. Suitable locks protect access to the structure.

```
typedef struct ph_feaddr_t{
      FPPI_FEID           feid;
      struct sockaddr_in  addr; /* IP Address of this FE */
```

```
}ph_feaddr;
```

This structure is the VIP header that is added to every packet sent to the FE, and stripped off of every packet received from the FE.

```
typedef struct ph_header_t
{
      FPPI_PortID portid;
      uint32_t    length;
} ph_header;
```

Part 7: FE Packet Handler for VxWorks

7 FE Packet Handler for VxWorks

The FE packet handler has no need of connecting to any form of VIDD. The design of the packet handler for VxWorks* is described below.

7.1 Design

The forwarding plane support for packet handling might be specific to the forwarding plane. In the PDK, the forwarding plane is IXA-based. Some of the main design goals for packet handling support have been the following:

4. **Operating system independence –** If the support for packet handling is not specific to the operating system on the forwarding plane, it becomes easier to port the same functionality to a different platform. The support can be provided on any platform that supports the IXA architecture.
5. **Use of simple OS primitives –** OS primitives like sockets and raw sockets are available on most BSD, Unix, and VxWorks platforms.
6. **No modifications to core components provided with the IXA platform –** This has been one of the main criterions for deciding how to perform packet handling. The current solution does not modify any of the existing core components or microblocks on the IXA platform.

7.1.1 Initialization

The Packet handler uses a VIP network socket (protocol 105) to send and receive packets to and from the CP. In order to do this, the packet handler needs to know the CP's IP address, and the `init` function will require it as a parameter.

Packets entering the FE from the network that are destined for the CP will enter the packet handler via the Forwarding Plane Module core component (FPM). How packets get to the FP module from the microblocks is of not in the scope of this document. The packet handler will register a callback with the FPM in order to receive these packets by calling the function `ix_cc_fpm_register_pkt_hdlr_cb`.

Packets coming from the CP destined for the network are received by a receiver thread that is reading packets from the above mentioned network socket. Therefore a new thread must be created by the init function.

7.1.2 Incoming Packets

Incoming packets follow the following sequence of steps as they traverse through the FE

1. The FPM receives packets from somewhere and passes them to the registered callback function. This callback is passed the bladeid, portid, length and buffer.
2. The packet handler adds a VIP header to the packet buffer, and sets the portid and length fields appropriately.

3. The packet is then sent out on the VIP socket (protocol 105).

7.1.3 Outgoing Packets

Outgoing packets are received from the CP, and passed on to the FPM. The following is the sequence of events:

1. The receiver thread starts reading from the network socket.
2. When a packet is read from the network, the IP and VIP headers are stripped off of the packet.
3. The portid is determined from the VIP header
4. The packet is given to the FPM through the function `ix_cc_fpm_sync_send_packet` along with the `portid`.