Operating System

Porting Guide

Control Plane-Platform Development Kit 2.11

March 2004

Contents

Figures

Part 1: Overview

1 Overview

The control plane PDK implementation was written initially for Linux*, which is a Posix-compliant operating system. Most modules are designed to be portable to any OS. The PDK uses the Platform Independence Layer (PIL) to ensure portability. The CP-PDK is ported and tested on VxWorks* and Linux platforms.

There are some components that contain platform-specific implementations or use platform specific libraries. This document explains what must be done to port these components to different platforms.

1.1 Terminology

Table 1 lists the terms used in this document and provides a definition for each term.

Table 1. Terminology

Term	Description
CP-PDK	Control Plane Platform Development Kit
Control Element (CE)	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)
Control Plane (CP)	See Control Element
FIB	Forward Information Base
Forwarding Element (FE)	In a separated control/data system, refers to the processor(s) responsible for fast path forwarding of data. Used interchangeably with FP.
Forwarding Plane (FP)	See Forwarding Element
IXA	Internet eXchange Architecture
NPU	Network Processing Unit
MPLS	Multiprotocol Label Switching
OS	Operating System
PIL	Platform Independence Layer
RCM	Route Cache Manager
VIDD	Virtual Interface Device Driver
OSAL	Operating System Abstraction Layer

1.2 References

Table 2 lists the documents that are referred in the OS porting guide document.

Table 2. References

Reference	Document Name
[1]	Route Cache Manager Design Reference
[2]	VIDD Design Reference
[3]	Forwarding Plane Plug-in Design Reference
[4]	Software Architecture Overview
[5]	Platform Independence Layer API Reference

Part 2: Platform Independence Layer

2 Platform Independence Layer

The Control Plane PDK uses the Platform Independence Layer (PIL) to enable portability. The PIL provides a platform-independent API for many common operations. It abstracts the common operating system services that the PDK uses. The PIL has been implemented and tested only for Linux, Win32* and VxWorks platforms. The PIL functionality should be ported to the new platform in order to port the PDK to other platforms. You can minimize the porting effort, as you do not have to port all the PDK code.

2.1 Porting Considerations

2.1.1 Support for Threads and Locks

The PDK is multi-threaded. The PDK uses the PIL functions to implement the thread and locking functions for ensuring uniform running of the PDK on all platforms. This ensures that the PDK code does not have to change on different operating systems. The implementation of the PIL functions for each operating system depends on the support provided by the OS for threads and locking.

For example VxWorks supports multiple tasks, but it does not support the concept of multiple threads. Therefore, the PIL thread functions for VxWorks use tasks to implement the thread functions, such as `PIL_CreateThread, PIL_GetCurrentThread, PIL_SuspendThread, PIL_WaitForThread,` and so on.

When porting the PIL to an unsupported OS, you must ensure that the PIL thread functions and the PIL critical section functions are fully supported.

2.1.2 Support for Simple IO calls

The PDK assumes support for the following:

- Simple IO function calls such as `read(), write(), select(),` and so on.
- Standard C libraries. If the standard C libraries do not support these simple IO calls, you should add such support in the PIL implementation.

Part 3: Operating System-Dependent Modules

3 Operating System-Dependent Modules

Some components of the PDK are OS-specific and are implemented using platform-specific libraries. The following sections describe these dependencies.

3.1 RCM

The Route Cache Manager (RCM) runs on top of the PDK and uses the IPv4 APIs to update the Forwarding Information Base (FIB) on the forwarding plane. The RCM acquires information on added or deleted routes from the kernel and invokes the IPv4 API for route table management.

The RCM module opens a routing socket using the RTNETLINK library to obtain the updated route table information from the kernel. The RCM module registers a callback function that is invoked when there is a change in the routing table.

RTNetLink is a user-level library that contains additional messages for kernel interaction, in addition to the standard NetLink messages. Refer to Section 1 for more information on NetLink and the specific functions used. . The RTNet library need not be supported on other platforms. If the RTNet library is not supported on other platforms, you must add additional functionality to complete the interaction with the kernel and to retrieve the route table information.

3.2 Virtual Interface Device Driver (VIDD)

The VIDD module simulates the forwarding plane interfaces on the control plane, and creates virtual interfaces for the user to manage. If a control plane management application changes the IP address of a forwarding plane interface, the VIDD module must reflect the change in the corresponding virtual interface. The VIDD module must reside in the kernel for this purpose. The VIDD is a kernel module in the PDK that is controllable from the user space.

The VIDD module is an OS-dependent layer, as it must reside in the kernel layer. It exposes IOCTLs to create and delete virtual interfaces. If you want to port the PDK to a new platform, you should rewrite the VIDD module to suit the new platform.

***Note*:** Newer versions of Linux (2.4.x) have built-in VIDD support and packet handler support. Therefore, there is no need to rewrite the entire driver and you can control it from the user space.

3.3 CE Packet Handler

The packet handler component performs the following:

- Deliversthe tunneled data packets from the FE to the corresponding virtual interface
- Looks for packets transmitted on any virtual interface and tunnels them to the correct FE for transmission on the correct egress interface.

The virtual interfaces that are created are treated as simple file descriptors in the current Linux implementations, and the CE packet handler module uses simple open, read, and select calls to look for packets and read the data.

Since the implementation of the VIDD is OS-dependent, the packet handler module should be changed depending on how the VIDD implementations treat the virtual interface. If the VIDD module is

implemented for virtual interfaces using a similar mechanism as Linux, there is no need to change the packet handler component.

For example, the implementation of the packet handler component will not change while upgrading to newer versions of Linux.

3.4 FE Packet Handler

The FP module uses raw and packet sockets to send the packets. The raw sockets are used for unicast packets and packet sockets are used for multicast packets. The raw sockets are more generic and are usually supported while the packet sockets are not universally supported in all operating systems. Since the packet-handling support is built into IXA SDK, none of the above two packets were used for version 1.1 of the PDK.

All locally destined packets are sent to the core for exception processing on an IXA platform. Additional support must be provided to distinguish between the packets that are to be sent to the local stack and the remote control plane. The exact nature of this support depends on the system being used and the stacks that are being run on the local and remote machines.

3.5 Forwarding Plane Module

The forwarding plane (FP) module of the PDK translates the NPF invocations to the specific Network Processor (NPU) invocations. The FP module consists of several FP plug-in managers that are responsible for specific functionalities. For example, the IPv4 FP plug-in manager is responsible for adding or deleting routes. The MPLS and QoS FP plug-in managers are responsible for implementing the MPLS and QoS functionalities of the router.

Each FP plug-in manager comprises a translator component and a platform-specific component. The translator components are fully portable across the operating systems and written using PIL.

The platform-specific component is specific to the NPU. The platform-specific component invokes some OS-specific network calls, such as setting the Layer 2 (L2) attributes of the interfaces, adding routes to the local IP stack on the data plane to synchronize with the NPU route tables, and so on. While porting across operating systems, you should consider the above factors of the platform-specific component, as the number of OS-specific calls is limited.

3.6 IXA SDK

The IXA SDK also uses an OS abstraction layer called the OSAL. The OSAL abstraction layer is different from the PIL. The PIL and OSSL layers may be merged in the future to create a generic OS abstraction layer that can be used across all components of the PDK.