ZebOS-XP®
Network Platform

Version 1.4
Extended Performance

Virtual Routing
Developer Guide

December 2015

IP Infusion Inc.
3965 Freedom Circle, Suite 200
Santa Clara, CA 95054
+1 408-400-1900
http://www.ipinfusion.com/

For support, questions, or comments via E-mail, contact:
support@ipinfusion.com

Contents

Preface

This guide describes the ZebOS-XP application programming interface (API) for Virtual Routing (VR) and Virtual Router Forwarding (VRF).

Audience

This guide is intended for developers who write code to customize and extend VR and VRF.

Conventions

Table P-1 shows the conventions used in this guide.

Table P-1: Conventions

Convention	Description
Italics	Emphasized terms; titles of books
Note:	Special instructions, suggestions, or warnings
`monospaced type`	Code elements such as commands, functions, parameters, files, and directories

Contents

This document contains these chapters and appendices:

- Chapter 1, *About Virtual Routing*
- Chapter 2, *Virtual Private Networks*
- Chapter 3, *Virtual Router Administration*
- Chapter 4, *Data Structures*
- Chapter 5, *Virtual Router API*

Related Documents

The following guides are related to this document:

- *Virtual Routing Command Reference*
- *Network Services Module Command Reference*
- *Network Services Module Developer Guide*
- *Installation Guide*

- *Architecture Guide*

Note: All ZebOS-XP technical manuals are available to licensed customers at http://www.ipinfusion.com/support/document_list.

Support

For support-related questions, contact support@ipinfusion.com.

Comments

If you have comments, or need to report a problem with the content, contact techpubs@ipinfusion.com.

CHAPTER 1 About Virtual Routing

This chapter provides an introduction to the Virtual Router (VR) feature implemented in ZebOS-XP.

Overview

VR logically subdivides a physical router into multiple virtual routers, allowing each virtual router to execute separate instances of the routing protocol and the network management software, for example, Simple Network Management Protocol (SNMP) or command line interface (CLI). Each virtual router can be independently monitored and managed by the user. Many sources refer to virtual routers in terms of their application within virtual private networks (VPNs). This design interprets the VPN implementation to be a specific application of the overall virtual routing design. As a result, the VPN approach is considered to be an add-on feature of virtual routing.

ZebOS-XP offers optional VR support for both IPv4 and IPv6. Virtual routing provides support for multiple Routing Information Bases (RIBs) and multiple Forwarding Information Bases (FIBs) per physical router. Each VR may consists of an Open Shortest Path First (OSPF) v2, OSPFv3, Border Gateway Protocol (BGP)- 4(+), or Routing Information Protocol (RIP), each with its own Route Information Base (RIB) and Forwarding Information Base (FIB).

In addition, the following IPv4 and IPv6 multicast protocol modules support Virtual Routing: Protocol Independent Multicast-Sparse Mode (PIM-SM and PIM-SMv6), and Protocol Independent Multicast-Dense Mode (PIM-DM and PIM-DMv6).

ZebOS-XP VR supports multiple instances of a routing table co-existing within the same router at the same time. Because the routing instances are independent, the same or overlapping IP addresses can be used without conflicting with each other. ZebOS-XP supports network devices that have the ability to configure different VRs, where each one has its own FIB that is not accessible to any other VR instance on the same device.

Physical Routers

Without virtual routers, a typical service-provider-to-customer relationship involves one physical router (PR) for services like VPN.

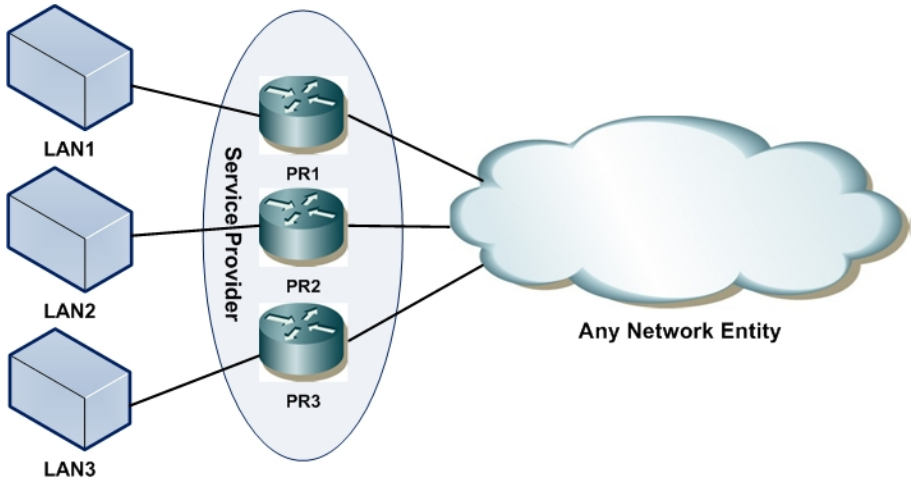


Figure 1-1: Service Provider with No Virtual Routers

In this example, each router functions as an extension of each customer's LAN.

Virtual Routers

Each virtual router (VR) has its own connection with the customer network, but may share physical resources, for example, several logical interfaces mapped to a physical router. A RedHat Linux implementation installs all routes in FIB 0 even when routes are installed in VR-specific FIBs.

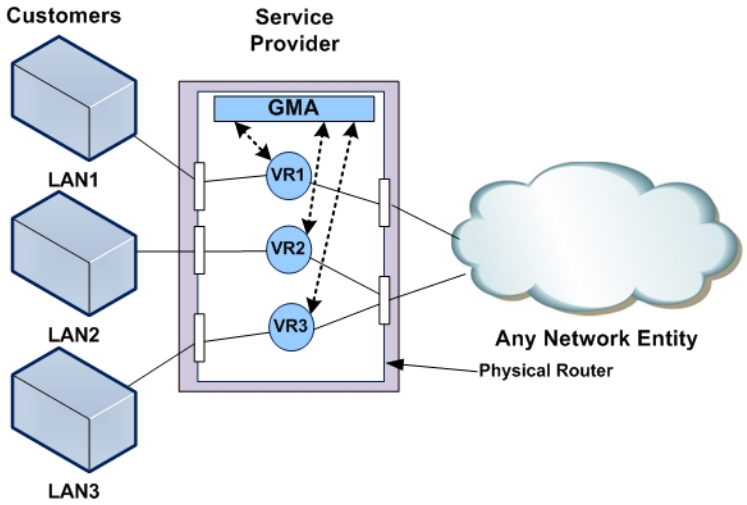


Figure 1-2: Service Provider with Virtual Routers

Note: IP Infusion Inc. recommends that VR 0 must be reserved for management purposes and not used for routing. All routing must be performed in the VR mode.

Architecture

A virtual router is a software emulation of a physical router. The many properties of a router are present in a virtual router: addressability, configurability, troubleshooting. All the tools used to configure the software in physical routers are available for virtual routers: CLI, scripts, telnet (remote configuration).

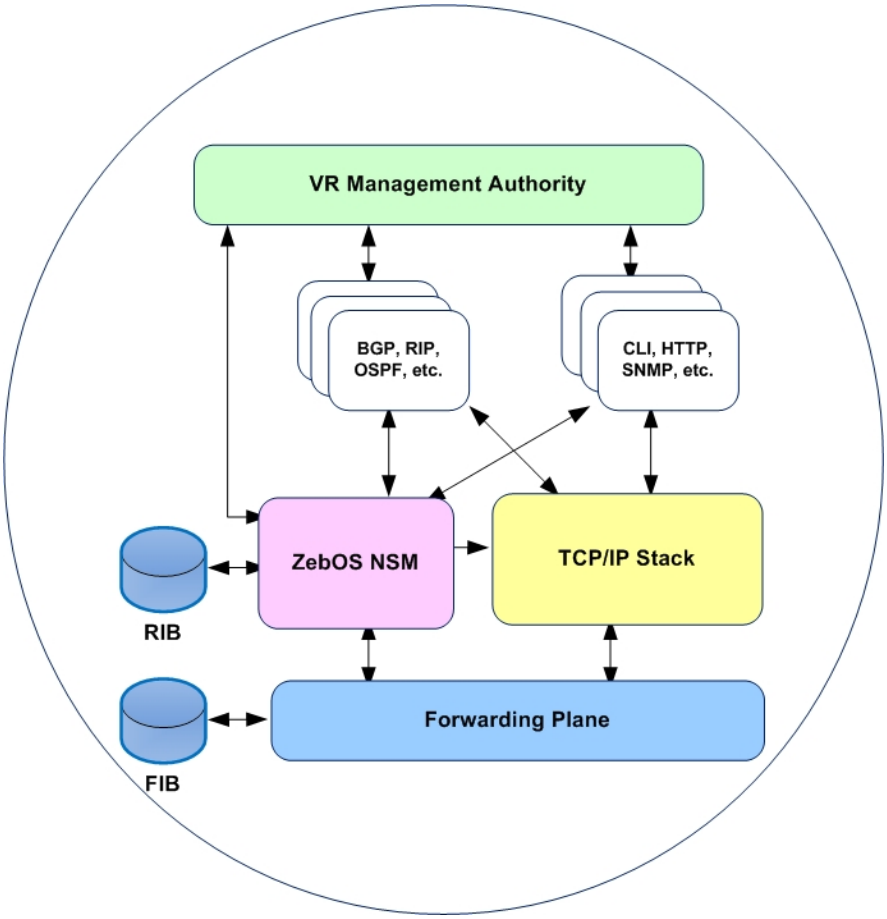


Figure 1-3: Virtual routers and other ZebOS-XP components

Each VR has a VR Management Authority (VRMA) that administrators use to configure and maintain the virtual router. Each VR also has a complete copy of the ZebOS-XP, with routing information base (RIB) and forwarding information base (FIB).

Virtual Router Forwarding

Virtual Router Forwarding (VRF) allows multiple instances of Routing Information Bases (RIB)s to co-exist within the same virtual router at the same time. Multiple VRFs inside the VR logically subdivide the routing tables. Each VRF is mapped to a FIB in a Linux namespace

Each VRF is mapped to a namespace in the Linux kernel. The default VRF in a default VR is bound to the default namespace. For all other VRFs, a new namespace is created and bound to the VRF.

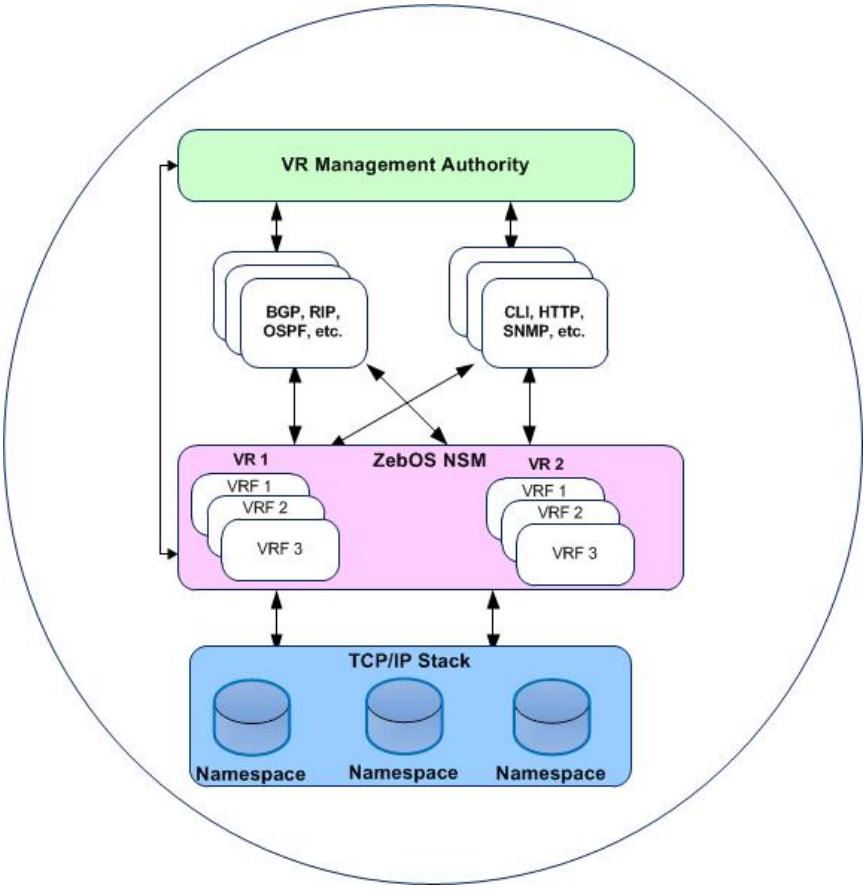


Figure 1-4: Virtual Routers Forwarding and Namespace

Global Management Authority (GMA)

Each virtual router handles its own physical connection configuration for the protocols it uses. The GMA manages allocation of physical router resources.

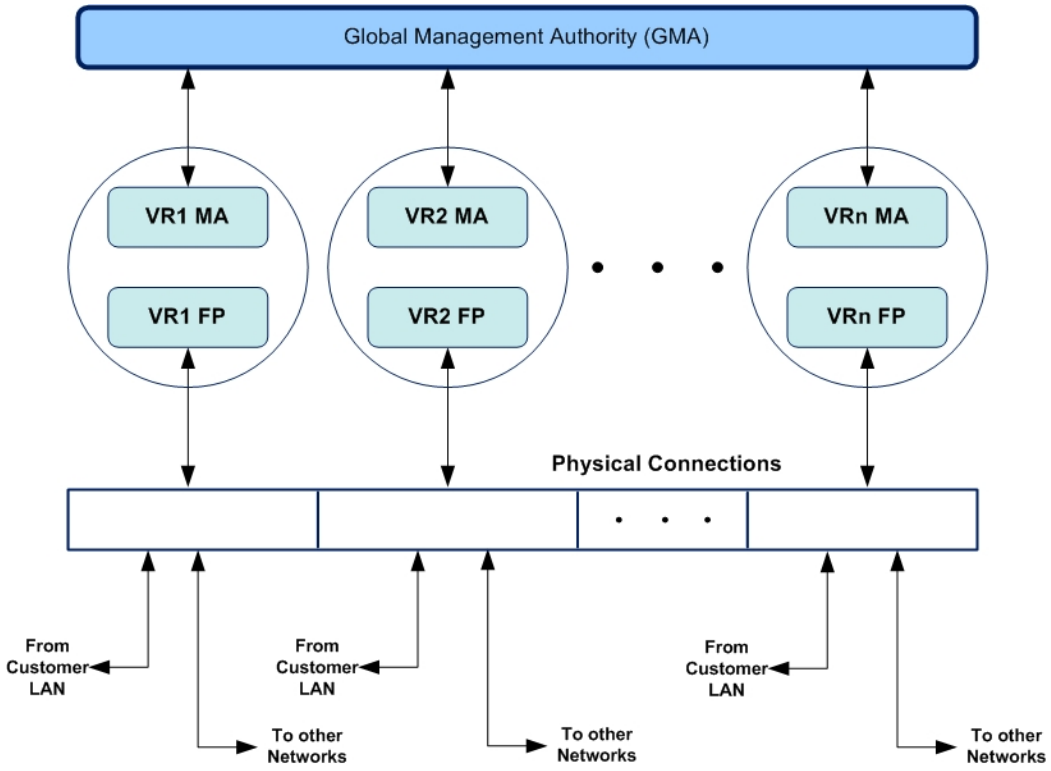
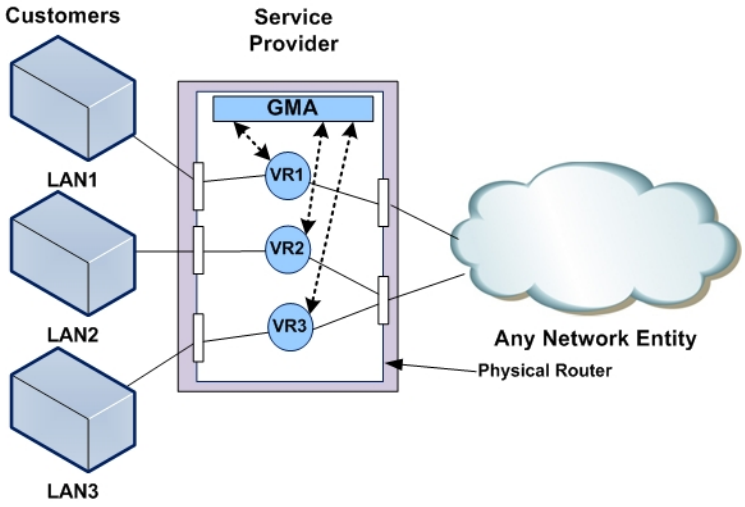


Figure 1-5: Global Management Authority for Virtual Routers

VRMA and GMA Interaction

The GMA resides in the router and provides services to each VR management authority (VRMA), but does not communicate on its own to resources outside the router.



Interactions with Other ZebOS-XP Components

The Integrated Management Interface shell (IMISH) interacts directly with the IMI module, which provides the service or passes the request for service to the Network Services Manager (NSM). IMISH offloads some processing from the NSM module. The Virtual Teletype Terminal (VTY) requires direct interaction with the NSM.

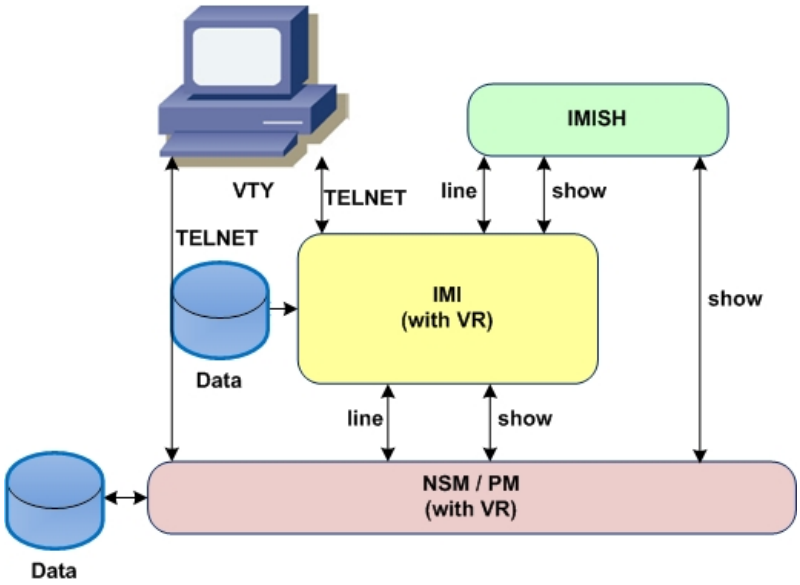


Figure 1-6: Basic Virtual Router Architecture including IMISH, IMI, NSM and VTY

Throughout the system, the Virtual-Router ID (VR-ID) must be tracked on a per-user basis. This permits the virtualization to be shielded from the user. The VR-ID is determined during login (based on the IP address that the user connects to), and is maintained by the VTY library or IMISH. Each VTY library or IMISH instance keeps the VR-ID during the session, and it is passed as the header of messages.

Note: In “line” messages, the reserved field is used. For “show” messages, no header is provided, so the VR-ID is simply encoded and decoded at the beginning of the message.

In addition to an IMISH connection, it is necessary to maintain VR-ID whenever messages are passed between the daemons for configuration get/set purposes.

VR CLI

In the VR CLI, the following two types of users are available:

- **Admin**—This user type is a global system administrator, who can:
  - Create and/or delete a Virtual Router on the system
  - Assign system resources
  - Configure a default Virtual Router in the system

  The admin user also has access to privileged commands described in the *Virtual Routing Command Reference* .
- **VR User**—This user type can configure the parameters assigned to their virtual routers. They are not able to configure or display parameters not assigned to their VR. VR users are not able to see resources beyond those assigned to their VRs. No privileged commands are available to a VR User.

With IMI/IMISH, login to a particular VR context is determined by the command line option for IMISH. The username and password are maintained per VR, and authentication is completed in each VR context. If the authentication fails, the user login is rejected.

Overlapped Address Spaces

An overlapped address space in the VR context is an IP address that is common to two or more physically separate networks. This support is enabled with the `--enable-vrf-overlap` flag in the configure script. For example, the network `10.10.0.0/24`, common to both VR1 and VR2, might be physically separate private networks, but it is not required to be. This means that VR1 and VR2 can both access the same network, or can access separate networks.

Because customers choose the addresses for their private networks, overlapping address spaces (private networks with identical addresses) are possible in the VR environment. To prevent VR protocols of such networks from accessing data and resources belonging to other VRs, the `--enable-vrf-overlap` option provides additional procedural checks and security.

CHAPTER 2 Virtual Private Networks

This chapter provides information about Virtual Private Networks (VPNs), an application of the Virtual Router (VR) support.

Overview

A VPN connects two geographically separate sites using a tunnel or predetermined route and encryption. A tunnel uses a router on each end with encryption/decryption software. VPN is one application that uses VRs to good advantage. With VRs described as emulations of physical routers, VPNs can be described as emulations of Wide Area Networks (WANs) that use Internet facilities to connect LANs.

When a customer wants to connect two sites using a service provider (an Internet Service Provider (ISP) or Plain old telephone service (POTS)), the service provider provides routing facilities as extensions to the customer's networks. To make a secure VPN, the service provider installs dedicated routers at each remote site.

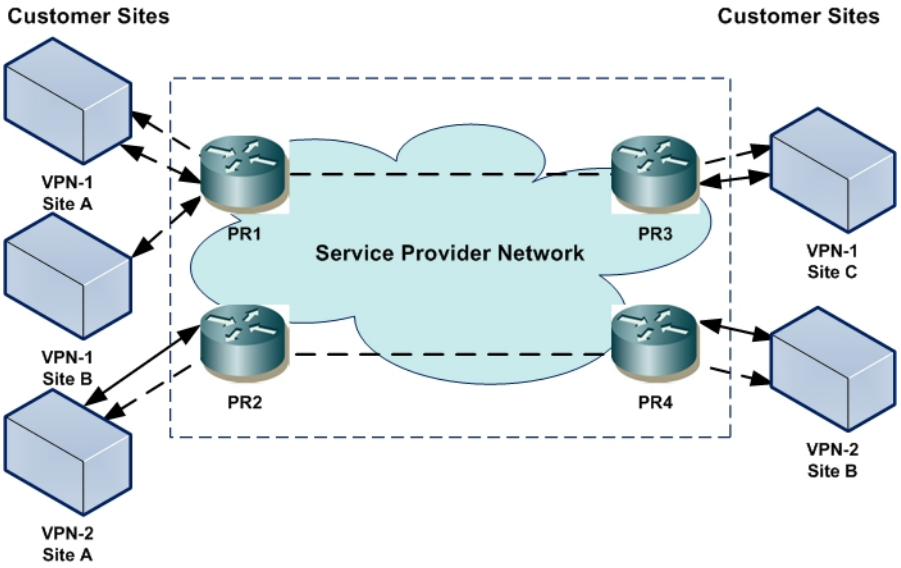


Figure 2-1: VPN without Virtual Routers

However, with virtual routing, the service provider instantiates VRs, as needed, in physical routers so that several customers share the same physical router as shown in the blue, vertical rectangles in Figure 2-2. Each customer gets a logically independent VR. The number of VRs that can fit on a single physical router varies with the number of routes, the mixture of running protocols, and physical limitations of the router (memory and processor speed). When sharing a physical router with several customers, capital costs to the service provider are reduced

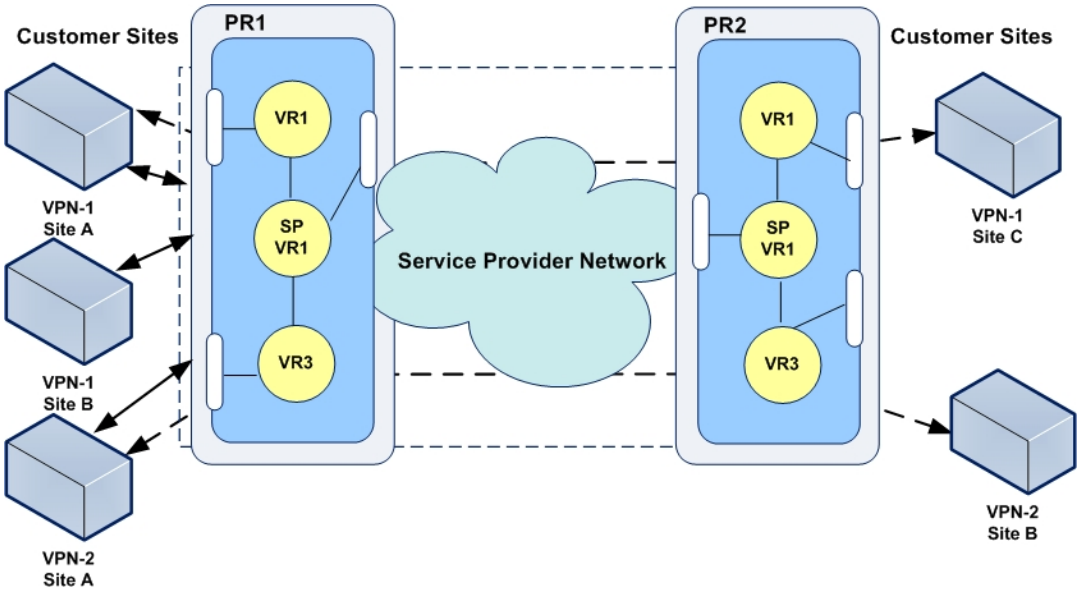


Figure 2-2: VPN with Virtual Routers

Note: An SP VR1 is a VR the service provider uses to provide additional access and functionality.

Sample VPN Application

- Two companies using the same service provider might have VPNs with different configurations. However, each VR contains a FIB, a RIB, a copy of ZebOS-XP, and copies of ZebOS-XP routing protocols
- Tunnels established between the pairs of customer sites
- Instances of Open Shortest Path First (OSPF) must also be active on each of the tunnels, which facilitates the exchange of reachability information between provider edge (PE) devices
- When VPN-1 sends a packet from 50.50.10.0/24 to 50.20.10.0/24, it is forwarded from VR-1 on PR1 to VR-1 on PR2 using a tunneling technique, then forwarded to the destination

A key assumption of the virtual routing design is that the VPN is already established. This is transparent to the routing protocols; they treat each tunnel end-point as a normal interface. OSPF operates in the same manner on a physical or virtual interface. The virtual router's management authority must know about these, however; and sets up, and tears down, within the VR itself. The management authority creates and destroys these virtual interfaces during startup, reconfiguration, and shutdown.

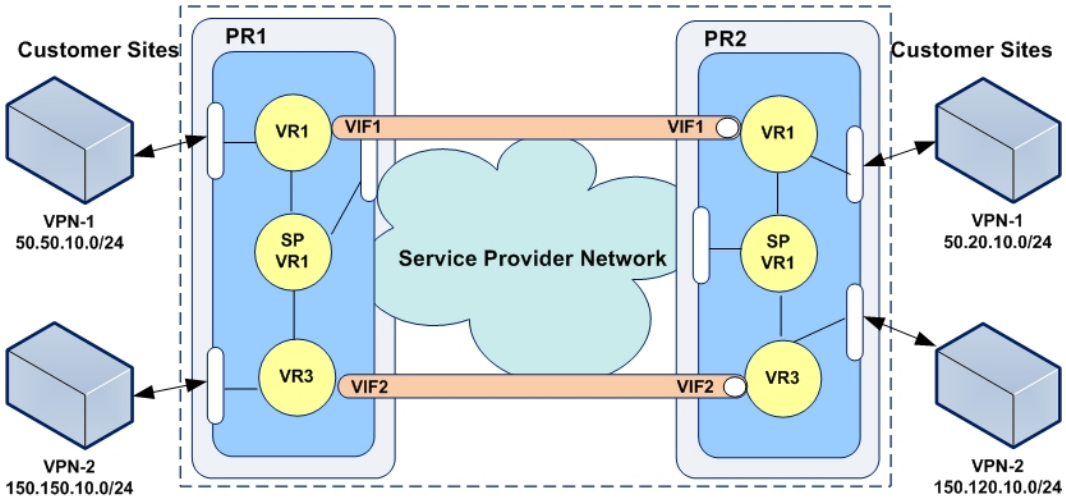


Figure 2-3: Virtual Routing Application

Note: Use a tunneling protocol capable of handling multicast packets; IP-in-IP tunneling is not sufficient for VR VPNs.

Service Provider Virtual Router—the Backbone Router

One of the key components of each physical router is the Service Provider Virtual Router (SPVR). The SPVR may play a number of different roles, depending on the deployment and architecture scenarios of the service provider.

In a scenario in which VRs are deployed over direct Layer-2 frame relay or ATM connections (or other) transport technology, the SPVR might not be utilized. When it is utilized, it might be limited to locating the appropriate tunnels and virtual interfaces for the individual VRs.

In a scenario in which multiple VRs are connected to the backbone through the SPVR, the SPVR provides functionality such as establishing tunnels, and passing appropriate information to the individual VRs; performing auto-discovery of other VRs; communicating with protocols such as Multiprotocol Label Switching (MPLS) for tunnel establishment; etc.

Any interfaces associated with the SPVR are used for connecting the VR tunnels, and reusing the routing table associated with the VR for tunnel communication. It is assumed that the lower layers handle the tunnel encapsulation.

TCP/IP Stack Implications

The TCP/IP stack used for the VPN solution supports the following:

- Logical (or virtual) interfaces
- Layering of logical interfaces (for tunneling purposes)
- Tunnel support for IP-IP, Simple Internet Transition (SIT), and Generic Routing Encapsulation (GRE) tunnels

Communication between NSM and the kernel for tunnel creation and manipulation is done via the `ioctl` function.

ioctl System Call

The `ioctl` system call allows a process to set or get different kernel parameters at the routing and interface levels. An `ioctl` call is performed via a socket, and usually looks like:

```
ioctl (socket, option, pointer)
```

where `pointer` points to memory for the kernel to read or write. This call is used throughout ZebOS-XP, mainly to create and manipulate tunnels. Table 2-1 lists the `ioctl` options supported by the kernel / IP stack onto which the VR-VPN feature is ported.

Table 2-1: ioctl options

Option	Comment
`SIOCGETTUNNEL`	Get information about a configured tunnel
`SIOCADDTUNNEL`	Create a new tunnel
`SIOCDELTUNNEL`	Delete an existing tunnel
`SIOCCHGTUNNEL`	Change the parameters for an existing tunnel

The third argument passed to the `ioctl` system call is the `ifreq` structure. All of the tunnel-specific information is stored in the structure `ip_tunnel_parm` and is defined in `linux/if_tunnel.h` file:

```
struct ip_tunnel_parm
      {
      char                    name[IFNAMSIZ];
      int                      link;
      __u16                    i_flags;
      __u16                    o_flags;
      __u32                    i_key;
      __u32                    o_key;
      struct iphdr             iph;
      };
```

The `ifreq` structure references this structure using code similar to the following:

```
struct ip_tunnel_parm p;
struct ifreq ifr;
...
ifr.ifr_ifru.ifru_data = (void *)&p;
...
```

VPN Tunneling

VPN is one of the significant applications of VR support (others include a private router for the customer at the ISP, and the ability to have multiple customers with potentially overlapping private networks immediately adjacent to the ISP). A simple way is to use VPN tunneling of some form (preferably the ability to use several tunnels at once within a single VR), with the endpoints looking to the routing protocols very much like point-to-point or point-to-multipoint connections (the latter, in the case of a tunnel with more than two endpoints). The endpoints are logical interfaces, and to a VR, are almost indistinguishable from a standard interface of the appropriate class.

The interface names that administrators choose can reflect the nature of the interface, for example, `interface virtual 1` instead of `interface fastethernet 1`. The routing protocols can treat these interfaces as if they were typical point-to-point or point-to-multipoint interfaces, regardless of the encapsulation method. The ability to set up and tear down VPN interfaces must be dynamic. There may be a requirement for a protocol that allows VPNs to discover each other. VPNs should not interfere with each other, even within the same VR, and must not interact in any way with each other across VRs. Routes that cross VPNs must operate normally. A routing option to drop packets with certain destinations, instead of using a default route for them, must also be provided. (It is not advantageous for packets destined for the other end of a VPN tunnel to be passed up the default route unencrypted — or at all — in case the VPN tunnel is down.)

NSM Implications

The ZebOS-XP VR uses the tunneling features of the IPROUTE2 utility package. These include creating and destroying tunnels, and forwarding between routers (or virtual routers) connected by these tunnels. These tunnels appear as point-to-point links to the different VRs, and support IP-in-IP (except multicast situations), SIT, and GRE types.

Using the IPROUTE2 utility from the command shell, this command establishes `t1` as an IP-in-IP tunnel:

```
ip tunl add t1 mode ipip remote 10.56.9.48 local 10.104.88.5
```

Tunnels can be established using the command shell before starting ZebOS-XP. When ZebOS-XP first starts, the `show interface` command lists the tunnels in the interface. They can then be assigned to a particular VR using the commands as described in the *Virtual Routing Command Reference*.

CHAPTER 3 Virtual Router Administration

Virtual router administration is divided between two entities: the Global Management Authority (GMA) that manages all VRs on a physical router; and the Virtual Router Management Authority (VRMA) that manages a single Virtual Router (VR).

GMA

The GMA is the component system administrators use to configure all VRs running on the same physical router. The tasks of the GMA include: allocating the resources of the physical router among the VRs, establishing the VRs, and granting access to the VRs to various user IDs.

VRMA

The VRMA is the component that customer network administrators use to configure a single VR. These administrators start and stop ZebOS-XP components, set up routes, and perform other router maintenance tasks.

The*Virtual Router Command Reference* contains a detailed description of the login procedure for VR administrative personnel.

Command Line Interface Limitations

Because there might be several VRs on any given physical router, ZebOS-XP enforces the restrictions described below.

VR Administrators

VR Administrators can:

- Any routing configuration task that applies to a physical router, including listing interfaces, setting up routes including: all data (static routes, host information, OSPF instance, and so on), and resources (interfaces) assigned to the VR.

VR Administrators cannot:

- Configure any resource not assigned to the VR
- List the interfaces of the physical router that do not belong, or are not assigned, to the VR
- Influence or access the RIBs and FIBs that belong to other VRs
- Access any GMA command or privileged function
- Start and stop ZebOS-XP daemons

GMA Administrators Architecture

GMA (Root) Administrators can:

- Create, delete, and maintain VRs, and assign interfaces to them
- Configure all system-wide attributes and parameters
- Move into, and configure, different VRs at any time
- Jump to a VR to perform maintenance tasks

GMA Administrators cannot:

- Jump from one VR to another without first jumping back to the GMA

For a complete list of commands reserved to administrators, see the *Virtual Router Command Reference*.

Note: The visibility of these commands for GMA Administrator is implemented with a privilege level. Whenever an `Admin` logs in to GMA context, Privilege Level 16 is granted. All these commands are installed as Privilege Level 16, therefore, VR users can not access these commands.

- `configure file`
- `description`
- `load`
- `load ipv6`
- `login virtual-router`
- `show running-config virtual-router`
- `show virtual-router`
- `username`
- `virtual-router`
- `virtual-router forwarding`

Multiple Concurrent Users

Because the Virtual-Router ID is tracked throughout the system during both configuration and show commands, multiple concurrent users are supported for the VR-CLI.

For each VR context (including GMA), only one user can enter the `Configure` mode. In order to support this functionality, it is necessary to store certain information when the NSM or PM commands return to the IMI after the configure command is issued. In particular, the cli->index value must be saved, because it is used in the subsequent commands for configuration. The value is saved under `struct host` as `void *cli_index` for both the IMI/IMISH and the VTY cases.

VR-ID Tracking

Throughout the system, the VR-ID must be tracked on a per-user basis. This permits the virtualization to be shielded from the user. The VR-ID is determined during login (based on the IP address to which user connects), and is maintained by the VTY library or IMISH. Each VTY library or IMISH keeps the VR-ID during the session, and the ID is passed as a header of the messages.

Note: In `line` messages, the reserved field is used. In `show` messages, a header is not provided, so the VR-ID is encoded/decoded onto the front of the message.

In addition to the IMISH connection, it is necessary to maintain VR-ID whenever messages are passed between the daemons for configuration get/set purposes.

Starting VR

The default location for the VR configuration file is `/usr/local/etc/VRNAME`.

Configuration Files

VR maintains the ability to load different VR configurations from different configuration files. In addition, the global configuration is read from a separate file during startup of IMI (or a PM if IMI is not available).

Case 1: IMI

If IMI is available, a single file is used per VR to store the configuration for all protocols.

Global Configuration File

Default file: `/usr/local/etc/ZebOS.conf`

Set file: Issue `-f` options when starting IMI process

Unset file: <none>

Read file: Read automatically on startup of IMI process

Write file: Issue `write file`, `write memory`, or `copy running-config startup-config` as root.

VR Configuration File

Default file: `usr/local/etc/<VRNAME>/ZebOS.conf`

Set using `configuration file default` under `VR` mode as Admin.

Set file: Use `configuration file WORD` under `VR` mode as Admin.

Unset file: Use `no configuration file` under `VR` mode as Admin.

Read file: Read automatically when VR is created.

Write file: Issue `write file`, `write memory`.

Case 2: Default CLI

If IMI is not available, each ZebOS-XP daemon (NSM, OSPF, BGP) has an individual configuration file on a per-VR basis.

Global Configuration File

Default file: `/usr/local/etc/nsm.conf`
`/usr/local/etc/ospfd.conf`
`/usr/local/etc/bgpd.conf`

Set file: Issue -f options when starting NSM/OSPF/BGP process

Unset file: <none>

Read file: Read automatically on startup of IMI process

Write file: Issue `write file`, `write memory`, or copy `running-config startup-config` as root.

VR Configuration File

Default file: `/usr/local/etc/<VRNAME>/nsm.conf` (for NSM)
`/usr/local/etc/<VRNAME>/PM.conf` (one for each PM)

Set file: <none>

Unset file: <none>

Read file: Read automatically when VR is created.

Write file: Issue `write file`, `write memory`.

A VR user does not have control over the configuration file used for their VR configuration because the user may or may not have visibility of, or permission to access, the file system on the device.

Configuration filenames are controlled via the Root mode of the VR-CLI.

CHAPTER 4 Data Structures

This chapter describes the data structures that are used in the Virtual Router (VR) functions.

Common Data Structures

The following data structures are common for all ZebOS-XP protocols and are used in VR functions:

- `interface`
- `lib_globals`

Refer to the *Common Data Structures Developer Guide* for a description of these data structures.

ipi_vr

This structure resides in the `lib/lib.h` file.

Member	Description
zg	Pointer to globals
name	Virtual router name
id	Virtual router identifier
router_id	Router identifier
flags	Flags whose values are: • LIB_FLAG_DELETE_VR_CONFIG_FILE • LIB_FLAG_VR_RE_CREATED • LIB_FLAG_VR_INACTIVE • LIB_FLAG_VR_NO_CONFIG
ifm	Master interface (of type if_vr_master)
vrf_vec	Virtual Routing and Forwarding (VRF) vector
vrf_list	VRF list
protos	Protocol bindings
host	Host (of type host)
access_master_ipv4	IPV4 access list (of type access_master)
access_master_ipv6	IPV6 access list (of type access_master)
access_master_generic	List of generic ACLs other than IPv4, IPv6 qualifiers

Member	Description
apbf_rule_group_master	Advanced Policy Based Forwarding (APBF) rule group master
prefix_master_ipv4	IPv4 prefix list
prefix_master_ipv6	IPv6 prefix list
prefix_master_orf	Orf prefix list
route_match_vec	Route match vector
route_set_vec	Route set vector
route_map_master	Route map master (of type route_map_list)
keychain_list	Key chain list (of type list)
proto	Protocol master
t_config	Configuration read event thread (of type thread)
vrf_in_cxt	VRF currently in context
t_if_stat_threshold	IF stats update threshold timer
entLogical	Ent logical
mappedPhyEntList	Mapped physical entity list
snmp_community	Community string to identify current virtual router
lib_vr_cdr_ref	Checkpoint Abstraction Layer (CAL) created record reference for library virtual router
pbr_rmap_event pbr_event	Policy Based Routing (PBR) event (of type pbr_rmap_event)

Definition

```
struct ipi_vr
{
  /* Pointer to globals. */
  struct lib_globals *zg;

  /* VR name. */
  char *name;

  /* VR ID. */
  u_int32_t id;

  /* Router ID. */
  struct pal_in4_addr router_id;

  u_int8_t flags;
#define LIB_FLAG_DELETE_VR_CONFIG_FILE    (1 << 0)
#define LIB_FLAG_VR_RE_CREATED            (1 << 1)
#define LIB_FLAG_VR_INACTIVE              (1 << 2)
```

```
#define LIB_FLAG_VR_NO_CONFIG             (1 << 3)

  /* Interface Master. */
  struct if_vr_master ifm;

  /* VRFs. */
  vector vrf_vec;

  /* VRFs. */
  struct ipi_vrf *vrf_list;

  /* Protocol bindings. */
  u_int32_t protos;

  /* Host. */
  struct host *host;

  /* Access List. */
  struct access_master access_master_ipv4;
#ifdef HAVE_IPV6
  struct access_master access_master_ipv6;
#endif /* def HAVE_IPV6 */

#ifdef HAVE_APBF
  /* Generic access-list master
   * List of generic ACLs other than IPV4, IPV6 qualifiers
   * For example src_mac, dst_mac, vlan etc
   */
  struct access_master access_master_generic;
  struct apbf_rule_group_list apbf_rule_group_master; /* APBF rule-group
                                                         master */
#endif /* HAVE_APBF */

  /* Prefix List. */
  struct prefix_master prefix_master_ipv4;
#ifdef HAVE_IPV6
  struct prefix_master prefix_master_ipv6;
#endif /* HAVE_IPV6 */
  struct prefix_master prefix_master_orf;

  /* Route Map. */
  vector route_match_vec;
  vector route_set_vec;
  struct route_map_list route_map_master;
  /* Key Chain. */
  struct list *keychain_list;

  /* Protocol Master. */
  void *proto;
```

```
  /* Config read event. */
  struct thread *t_config;

  /* VRF currently in context */
  struct ipi_vrf *vrf_in_cxt;

  /* If stats update threshold timer */
  struct thread *t_if_stat_threshold;

  struct entLogicalEntry *entLogical;
  struct list *mappedPhyEntList;

  /* Community string to identify current VR */
  struct snmpCommunity snmp_community;

#ifdef HAVE_HA
  HA_CDR_REF lib_vr_cdr_ref;
#endif /* HAVE_HA */

#ifdef HAVE_PBR
  struct pbr_rmap_event pbr_event;
#endif /* HAVE_PBR */
};
```

CHAPTER 5 Virtual Router API

This chapter describes the API used for ZebOS-XP virtual routers (VRs).

Overview

The VR library can be linked in many ways and different sets of VR features are available depending on the type of initialization that is performed.

The VR implementation relies on centralized control over the data so that virtual routers and other resources can only be created or deleted from a central location. When IMI is available, this is performed through IMI; if IMI is not available, it is performed through NSM.

Note: Certain configuration is always controlled by NSM.

To allow the VR library to generically interact with the ZebOS-XP component with which it links, a set of API callbacks are provided. Functions in the `/vr` directory provide the generic top-level configuration, and the details are performed by the daemon process if a callback has been registered.

Some of these APIs are also used to get or set the protocol-specific configuration associated with virtual routing.

The VR API callbacks are defined in the file `/lib/lib.h`.

```
enum vr_callback_type
{
  VR_CALLBACK_ADD,
  VR_CALLBACK_DELETE,
  VR_CALLBACK_CLOSE,
  VR_CALLBACK_UNBIND,
  VR_CALLBACK_CONFIG_READ,
  VR_CALLBACK_ADD_UNCHG,
  VR_CALLBACK_MAX
};
```

The API callbacks are registered using the `ipi_vr_add_callback` function:

```
void
ipi_vr_add_callback (struct lib_globals *zg, enum vr_callback_type type,
                     int (*func) (struct ipi_vr *));
```

VR API

This section describes the `ipi_vr_add_callback` function used for VR.

ipi_vr_add_callback

This function sets the callbacks to virtual router based on the vr_callback_type. The vr_callback_type are defined in `/lib/lib.h`.

Syntax

```
void
ipi_vr_add_callback (struct lib_globals *zg, enum vr_callback_type type,
                     int (*func) (struct ipi_vr *))
```

Input Parameters

`zg` library global structure

`type` VR API callback types:

- `VR_CALLBACK_ADD`
  Sets the VR add callback function
- `VR_CALLBACK_DELETE`
  Sets the VR delete callback function
- `VR_CALLBACK_CLOSE`
  Sets the VR close callback function
- `VR_CALLBACK_UNBIND`
  Sets the VR unbind callback function
- `VR_CALLBACK_CONFIG_READ`
  Sets the configuration read callback function
- `VR_CALLBACK_ADD_UNCHG`
  Sets the add unchanged callback function
- `VR_CALLBACK_MAX`
  Represents the MAX value

`function pointer`
Pointer to a function to call when one of the events that corresponds to a vr_callback_type takes place

Output Parameters

None

Return Value

RESULT_OK when the function succeeds

PAL API for VR

The PAL functions are interface-specific to the PAL layer for VR.

Each operating system has its set of PAL functions defined in "pal/*operatingsystemname*/pal_socket.c" file where *operatingsystemname* stands for the name of a given operating system. For example, the `pal_sock_get_bindtofib` function used in Linux is defined in "pal/linux/pal_socket.c".

Function	Description
pal_kernel_fib_create	Creates a forwarding information base (FIB) in the forwarding plane for a given FIB ID
pal_kernel_fib_delete	Deletes a FIB in the forwarding plane for a given FIB ID
pal_kernel_if_bind_vrf	Binds an interface to a VR in the forwarding plane
pal_kernel_if_unbind_vrf	Unbinds an interface from a VR in the forwarding plane
pal_sock_get_bindtofib	Gets the FIB ID associated with the VR for a given socket file descriptor
pal_sock_set_bindtofib	Binds a socket to a VR in the forwarding plane

pal_kernel_fib_create

This function creates a FIB in the forwarding plane for a given FIB ID.

Syntax

```
result_t
pal_kernel_fib_create (fib_id_t fib, bool_t new_vr, char *fib_name)
```

Input Parameters

`fib`	FIB identifier
`new_vr`	Flag that indicates if new VR is being created
`fib_name`	Name of the FIB

Output Parameters

None

Return Value

RESULT_OK when the function succeeds

RESULT_ERROR when a generic error occurs

pal_kernel_fib_delete

This function deletes a FIB in the forwarding plane for a given FIB ID.

Syntax

```
result_t
pal_kernel_fib_delete (fib_id_t fib)
```

Input Parameters

`fib`	FIB identifier

Output Parameters

None

Return Value

RESULT_OK when the function succeeds

RESULT_ERROR when a generic error occurs

pal_kernel_if_bind_vrf

This function binds an interface to a VR in the forwarding plane.

Syntax

```
result_t
pal_kernel_if_bind_vrf (struct interface *ifp, fib_id_t fib_id)
```

Input Parameters

`ifp`	Pointer to interface structure
`fib_id`	FIB identifier

Output Parameters

None

Return Value

RESULT_OK when the function succeeds

RESULT_ERROR when a generic error occurs

pal_kernel_if_unbind_vrf

This function unbinds an interface from a VR in the forwarding plane.

Syntax

```
result_t
pal_kernel_if_unbind_vrf (struct interface *ifp, fib_id_t fib_id)
```

Input Parameters

`ifp`	Pointer to interface structure
`fib_id`	FIB identifier

Output Parameters

None

Return Value

RESULT_OK when the function succeeds

RESULT_ERROR when a generic error occurs

pal_sock_get_bindtofib

This function gets the FIB ID associated with a VR for a given socket file descriptor.

Syntax

```
result_t
pal_sock_get_bindtofib (pal_sock_handle_t sock, fib_id_t * fib)
```

Input Parameters

`sock`	Socket file descriptor

Output Parameters

`fib`	FIB identifier

Return Value

RESULT_OK when the function succeeds

RESULT_ERROR when a generic error occurs

pal_sock_set_bindtofib

This function binds a socket to a VR in the forwarding plane.

Syntax

```
result_t
pal_sock_set_bindtofib (pal_sock_handle_t sock, fib_id_t fib)
```

Input Parameters

`sock`	Socket file descriptor
`fib`	FIB identifier

Output Parameters

None

Return Value

RESULT_OK when the function succeeds

RESULT_ERROR when a generic error occurs

Index