ZebOS-XP®
Network Platform

Version 1.4
Extended Performance

Bidirectional Forwarding Detection
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 Bidirectional Forwarding Detection (BFD).

Audience

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

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 guide contains these chapters:

- Chapter 1, *Introduction*
- Chapter 2, *BFD Software Architecture*
- Chapter 3, *BFD Application Protocol Modules*
- Chapter 4, *Data Structures*
- Chapter 5, *OAM for BFD and MPLS*
- Chapter 6, *Network Services Module*
- Chapter 7, *Border Gateway Protocol*
- Chapter 8, *Intermediate System To Intermediate System*
- Chapter 9, *Open Shortest Path First*
- Chapter 10, *Routing Information Protocol*
- Chapter 11, *BFD Authentication*
- Chapter 12, *Virtual Router*
- Chapter 13, *BFD Command API*

- Chapter 14, *BFD MIB Support*

Related Documents

The following guides are related to this document:

- *Bidirectional Forwarding Detection Command Reference*
- *Bidirectional Forwarding Detection Configuration Guide*
- *Network Services Module Developer Guide*
- *Network Services Module Command Reference*
- *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 Introduction

The Bidirectional Forwarding Detection (BFD) module supports BGP, IS-IS, OSPFv2, and RIP protocols as part of the ZebOS-XP protocol suite.

Protocols often rely upon a relatively slow “Hello” mechanism to detect failures when there was no hardware signaling to assist. Detection times in protocol modules are often no better than a few seconds. This delay is too long for critical applications and represents a loss of data at very high processing rates. BFD works in conjunction with BGP, IS-IS, OSPFv2, and RIP to detect failures faster.

Features

BFD also provides the following features

- Low-overhead, short-duration detection of failures along the path between adjacent forwarding engines
- Rapid detection of communication failures between adjacent systems to more quickly establish alternative paths
- A single mechanism to detect liveliness over any media and in any protocol layer
- Passive, Active, Synchronous, Asynchronous, and Demand modes of operation
- Improved system performance when faster detection is required, because data-plane reachability detection is detached from control-plane functionality
- Sub-second detection times, similar to those provided in SONET/SDH networks
- A separate process in ZebOS-XP works in conjunction with routing protocols and NSM to detect forwarding plane reachability to protocol next hops
- Protocol modules support BFD irrespective of where BFD packet-sending operations take place; in the interfaces, data links, or to some extent, in the forwarding engines themselves
- BFD is Graceful-Restart unaware. Whenever BFD timers expire, a session-down event is triggered to the protocol module, and BFD maintains sessions for the protocol while it undergoes Graceful Restart
- BFD state machine interactions, as defined in the IETF drafts, are supported
- A faster mechanism to detect liveliness of static next-hops

Operation

BFD only works when both ends of the connection support BFD, making it incrementally deployable. With BFD, faster detection times are possible without overloading the control-plane CPU, thus allowing it to focus on other control plane tasks. Because it enhances control plane protocol performance, BFD should be run at all times. When BFD is running, it provides critical functions, so timer values must be correctly configured. This is because very small timer values can cause flaps and large values can cause BFD detection to become redundant.

If a BFD failure occurs, the time required for the backup system to come online can be just milliseconds, based on the data plane failure detection time desired.

Failover Modes

BFD may be run in two failover modes:

- It shares the fate of the control plane.
- It does not share the fate of the control plane.

In non-fate-sharing mode, BFD is Graceful-Restart friendly, because it allows the data plane to run independently of the control plane. In fate-sharing mode, BFD may inhibit protocol Graceful-Restart mechanisms, due to its shorter detection periods. Although the ZebOS-XP BFD module can be run in both modes, owing to the ZebOS-XP architecture, it runs only in a “non-fate-shared” mode.

Design

Each protocol module that supports BFD uses BFD client APIs to communicate with the BFD base module. These APIs are connected to and exchange information with the BFD server module. The BFD server module encodes and decodes messages from clients and configures sessions in the BFD base module. Using the BFD abstraction layer, the base module configures relevant information for the BFD Hello processing module. In turn, the BFD Hello module does all packet processing and FSM-related functions. Session triggers and data plane-related information are passed by the Hello processing module back to the BFD base module via the abstraction layer. A BFD abstraction layer is supported to allow sessions to configure remotely when BFD is used in a distributed architecture.

Mode of operation

BFD is supported in a monolithic architecture. BFD process runs on the same card (which may be the control card) along with other ZebOS-XP protocols. The BFD Abstraction Layer can be a dummy layer. The BFD Base Module directly calls functions in the BFD Packet processing module. The BFD Abstraction layer is a thin layer that provides abstractions and makes various direct function calls.

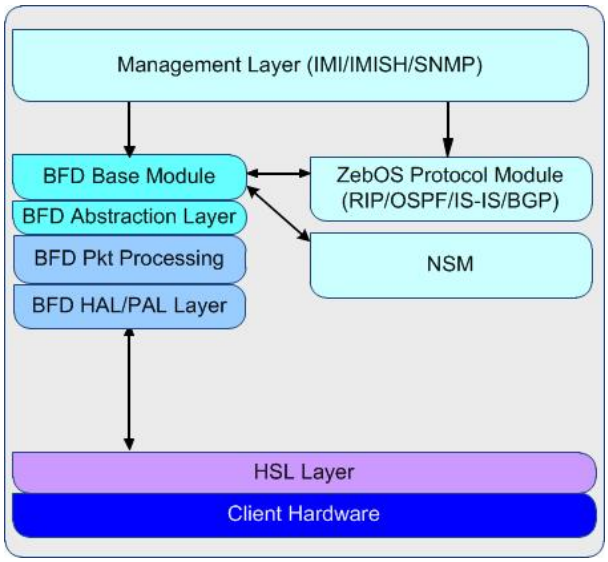

Figure 1-1: BFD Mode of Operation

The HAL/PAL module attached to the BFD module in turn sends BFD packets to the hardware.

CHAPTER 2 BFD Software Architecture

The Bidirectional Forwarding Detection (BFD) module works with most router architectures wherever hardware supports some level of BFD capabilities, and in situations where there is no hardware support at all. The BFD module is designed to work in conjunction with application protocol modules (for example, OSPF, BGP, RIP) to enable them to configure BFD sessions and for the sessions to get the bidirectional forwarding failure notifications from BFD. The way each application reacts to a session-down event is application-specific.

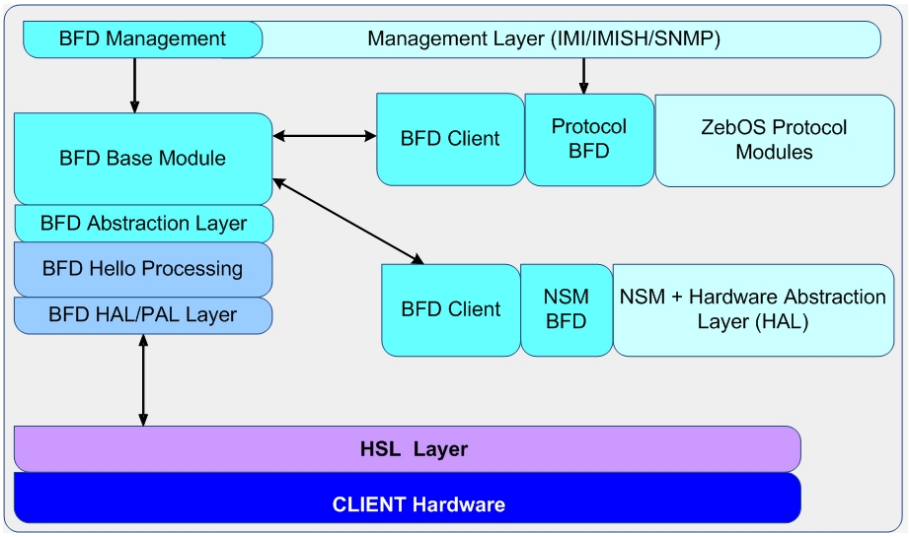

Figure 2-1: BFD Software Architecture

BFD Base Module

The BFD base module:

- contains most of the functionality as defined in the BFD base specification. All protocol modules and the NSM interact with the base module using the BFD client to program BFD for related functionality. Session information as configured by the application resides in the base module.
- interacts with management modules, and is the data store for all relevant information configured by management. It stores all protocol-specific information and configures forwarding session-specific information to the Hello sending module.
- interacts with the NSM to get interface and VR/VRF-specific information. However, BFD itself does not use VR or VRF information.
- contains the counter-polling mechanism that retrieves current counter values from the BFD forwarding module for BFD forwarding-state-related counters. Counters clear-on-read counters and the polling interval is 5 seconds by default.
- receives forwarding -elated information from the BFD Hello Processing Module, including the trigger for session Up/ Down and forwarding-related information.
- is hitless-restart aware. A BFD session is not terminated, even when the BFD protocol module goes down. The BFD base module maintains the session for the period configured by each protocol or session. However, if the BFD session goes down, it silently stops the session.

- tracks all interface-specific and multi-hop BFD information.

Sessions are identified by source address, destination address, client module identifier, interface index of the port, lower layer, and whether the session is multi-hop or not, for the applications. For protocol and management purposes, a session is identified by a system-wide local identifier that is unique across an individual system (or box).

BFD Abstraction Layer

The BFD Abstraction Layer abstracts the forwarding module from the base BFD functionality: In other words, the hello-sending process is abstracted from the base BFD module. Whenever a global parameter is changed in the base module, the information is propagated to the hello-processing module via the line card.

The abstraction layer also abstracts the (reverse data flow) Hello-sending module to the BFD base module. This implies that all session triggers to the BFD base module pass from the BFD hello layer through the abstraction layer.

All data plane functionality, such as counters, are polled by the base module through the abstraction layer.

BFD Hello Processing

The Hello Processing module processes hellos received from a neighbor and sends hellos when a session is configured from the BFD base module. This module keeps track of packet statistics and other forwarding plane functionality. All shadow session information required for forwarding is maintained by this module.

The hello-processing module can be co-located with the BFD process in a monolithic architecture (HAVE_BFD_MONO), and in the hardware or software on a different card or with a different process in a distributed architecture (HAVE_BFD_MAIN or HAVE_BFD_LINE).

BFD hello processing triggers BFD session Up/ Down events in the base module. In hardware, the Hardware Services Layer/Hardware Abstraction Layer (HSL/HAL) polls for the session Up/Down message, then notifies BFD appropriately.

BFD Client Module

The BFD client is a library attached to all application protocol modules, the NSM, and other modules that require the services of the BFD base module. The BFD client module is responsible for encoding and decoding messages between the BFD base module and a client application, such as NSM or a ZebOS-XP protocol module. The BFD client in turn calls appropriate functions registered, based on the message received from the BFD base module. This module configures client-specific session information such as session add, session delete and session modify. Information also includes Graceful Restart-related information.

The BFD client is triggered by a BFD server for a session Down/Up or a BFD protocol Down/ Up, in which case it submits information to the respective protocol module. The BFD client layer propagates various BFD messages to the protocol module identified by the BFD message module. It follows the NSM client module design in that it abstracts the BFD message module to the protocol module. The main tasks of the client module are:

- BFD message communication management
- BFD session management
- Graceful restart management

First, the client module maintains BFD message communication. It initiates a BFD message to the BFD server in the initialization phase, and ensures that the connection is up and running until termination. In the event of communication loss, the client retries initiation to the BFD server at specific intervals. This avoids dependency on protocol module startup order.

Second, the client module maintains a BFD session to multiplex BFD messages coming from the BFD server to a protocol module. This is necessary because certain protocol modules require an identical BFD session for different entries, for example, the NSM module for static route next-hop detection. Overall performance is improved for multiple-entry lookup, and BFD messages are minimized between the client and server by avoiding the duplicate messages for the same session. Finally, the module takes care of graceful-restart. It synchronizes BFD session information with the BFD server in case of planned or unplanned graceful-restart events. It automatically detects the server existence by re-connecting to the server periodically and re-issues BFD session requests to the server once it connects.

Client-Server IPC

BFD solution provides a modularized architecture by separating protocol-specific functions (BFD client) from the core BFD module (BFD server). This ensures that CPU-intensive BFD operations do not affect the main task of a protocol module. Communication between the BFD server and the BFD client is handled through a standard client-server IPC. BFD server creates a socket and waits for the connection requests from the BFD clients. BFD clients, on the other hand, initiate a connection during the start-up phase, and make sure that the connection is up by doing a connection retry just in case the BFD server is not up and running. After connection set-up, the BFD client issues BFD message (session add and the session delete) to the BFD server so that it starts track reachability. When the BFD server identifies a state change in a BFD session, for example, session up or session down, it propagates this to BFD clients via the BFD client-server IPC. The IPC is closed when either the BFD client or server terminates their process.

Client and Server Restart Scenario

The ZebOS-XP BFD module supports BFD server and client restart scenarios. Here is how:

BFD Server Restart

In the case of BFD server restart, for example, the BFD server process is upgraded, BFD clients retry the IPC connection until the server comes back up and accepts the connections from the client. Once connections are established, clients will re-issue BFD session requests to the server in a batch process so that the BFD server starts maintaining BFD sessions for clients, just as before the restart took place.

BFD Client-Server Restart

In the case of BFD client restart, for example, OSPFv2 graceful restart, a BFD client supplies restart parameters (such as grace period) to the BFD server beforehand, if it is able to do so. When a client terminates the process, the BFD server maintains a BFD session for the client. Clients can re-issue BFD session requests to the server after restart occurs, just like the normal start process, and override previously requested sessions maintained by the server during the restart period. The BFD server flushes the sessions that are not overridden after the grace period expires.

BFD Server Module

The BFD server module manages BFD message handling on the BFD server side with the following tasks:

- BFD message communication management
- BFD client management
- Graceful restart management

First, it accepts a BFD message communication request from a protocol module by opening a streaming socket, either a Unix domain socket (default) or a TCP socket. It follows same approach to NSM message communication using a different Unix path (/tmp/.bfdserv) or port (TCP/4600). Second, it maintains BFD clients to correctly propagate BFD session messages to the client, and avoids duplicate client connections from the same protocol module. Finally, it synchronizes BFD session information during graceful restart, both for the protocol modules and the BFD server.

BFD and NSM

The NSM BFD module is responsible for BFD-specific interactions required in NSM, including checking the connectivity to a static route next hop, which is configured in NSM. This module is similar to a protocol client module. The difference between NSM and other protocols is that, unlike protocols that set a trigger to BFD when a particular neighbor entity is discovered, NSM static routes have no protocol-specific information. Therefore, sessions for NSM static routes need to be persistent.

Note: BFD support for NSM static routes is not available in ZebOS-XP.

NSM BFD Module

This is the plug-in module for NSM that supports BFD capability for both IPv4 and IPv6. It manages the following tasks in addition to initialization and termination of the BFD client module:

- Callback registration for the BFD session-event handling
- Session initiation towards BFD in OAMD for static-route next hops
- Static-route function handling calls based on the callback trigger received from BFD

Currently, NSM applies BFD as the nexthop reachability-detection method only for IPv4 static routes. It adds a BFD session when NSM creates a static route entry in the NSM RIB, and deletes the session when NSM removes the static route from the NSM RIB.

A BFD client session key is generated with a combination of outgoing interface index as the BFD interface index, the IP address of the outgoing interface corresponding to the static route, and the nexthop IP address as the BFD destination address. A persistent session flag is set for all sessions, and the multi-hop flag is set if the nexthop address is not in the same subnet as the outgoing interface address. A single BFD session is initiated for all static routes configured on a specified interface, using, or resolving to, a designated directly-connected nexthop address. In other words, for multiple static routes using the same nexthop address through a particular interface, only a single BFD session exists.

Callback functions are registered so NSM can respond to BFD session events, for example, session up, or session down. Enabling or disabling the static route feature on an interface basis or globally, on all interfaces, triggers the creation or deletion of BFD sessions on a specified interface. When all static routes for a given interface and nexthop are removed from the configuration, only then is the corresponding BFD session deleted. The ADMIN_DOWN flag for the BFD session-down request is set if the session-down request is triggered by an operator command.

BFD HAL/PAL Layer

The BFD HAL/PAL layer is an API layer exposing the APIs to send BFD packets over hardware with appropriate encapsulations. This layer currently serves to send BFD packets out of an interface in a session to a peer. Since there is no support in the hardware module, the PAL module is the only one currently populated and defined.

Admin Down State

BFD allows an administrative down (admin down) state on each interface for a single-hop session, or on a neighbor basis for a multi-hop session. In this state, hellos are sent with the session state to admin down. This allows the peer to bring down the BFD sessions without bringing down the client session.

Echo Design

BFD Echo mode is configurable on a per-interface basis for all single-hop BFD sessions. BFD echo mode is not defined for multi-hop sessions. To send a packet in echo mode a session must be in the Up state. Once the session is up, an echo packet is sent to the reserved BFD echo port addressed to its own address at the IP layer and the peer at the link layer. This tests the actual forwarding path. Echo mode can be active in both demand and asynchronous mode.

Session FSM Module

The BFD FSM is maintained in the BFD Packet Processing module. A session can be in any of the following states:

Admin Down. The session has been administratively brought down. .

Down. The session is in the Down state; no valid packet from the neighbor has been received. .

Init. This state is reached when a packet from a neighbor has been received, however the neighbor lists the session in. the Down state, so a three-way handshake is not complete.

Up. This state is reached when both the ends of the session have received valid packets from the neighbor and the . three-way handshake is complete.

BFD Message Module

The message module manages BFD client and server communication. It follows the same design principle as the NSM client and server communication for these reasons:

- Robustness
- Extensibility
- Reliability
- Portability

Robustness of the NSM message module is one of the key features of the ZebOS-XP modularized architecture. The BFD message module also uses the TLV message format, like NSM, so it is also fully extensible. The underlying IPC mechanisms, Unix domain socket (default) or TCP, is also common to NSM, thus guaranteeing message delivery and portability.

Interfaces

The BFD module interacts with external modules, as shown in the diagram that follows. The main interfaces are to:

- Management (SNMP/CLI)
- Application Protocol(s)
- BFD Distribution Layer
- NSM APIs
- HAL/PAL APIs

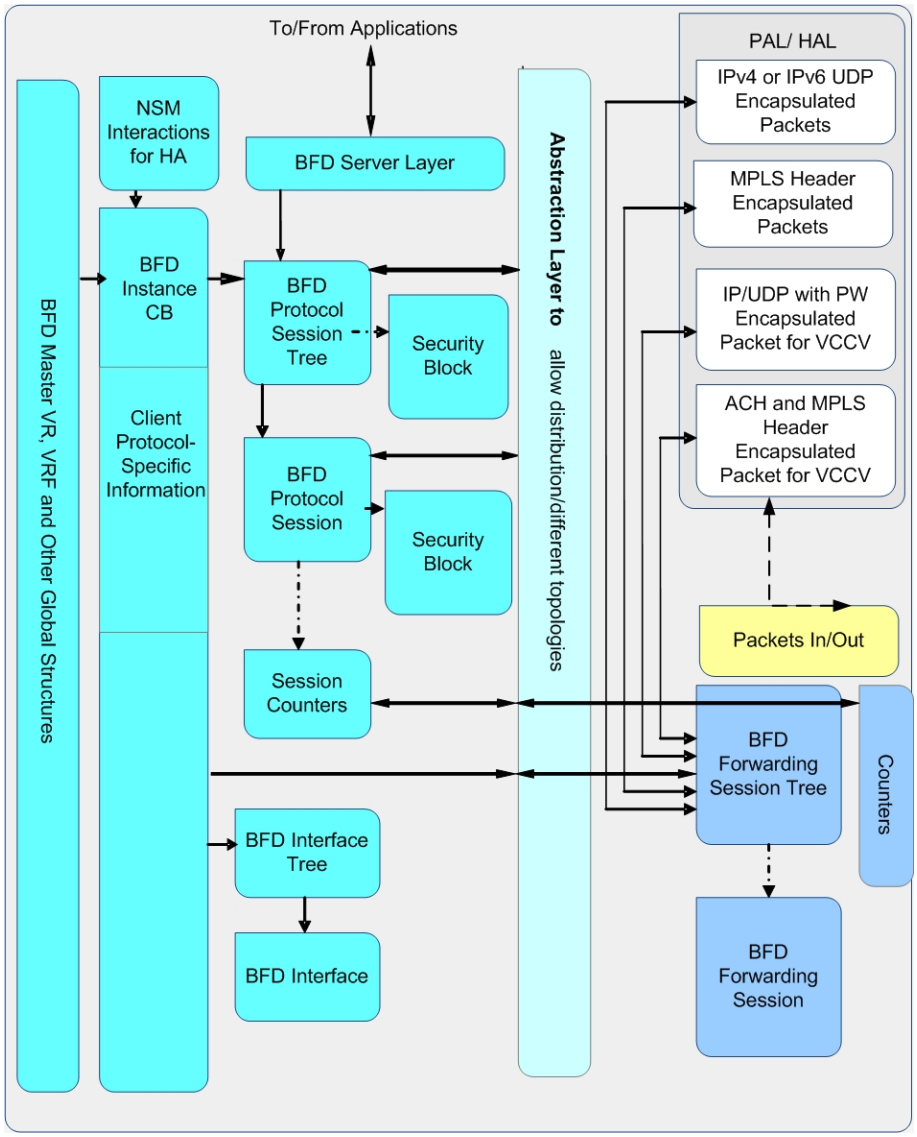

Figure 2-2: BFD Module interfaces

Management Interface

The BFD Management module consists of the CLI modules. A management interface module is required to program BFD-related parameters in the BFD module. Typical operations are to set various timer values, and the BFD mode, whether echo or asynchronous, on a router or interface basis. The management interface is also used to set the parameters for multi-hop BFD sessions.

Application Protocol Interface

The BFD Application Protocol Interface (API) provides the interface to the application protocols requiring forwarding plane liveliness detection. Typical operations are to create, delete, or update a new session. Another operation is to specify BFD Graceful Restart parameters, if any, for a session.

Distribution Interface

The BFD Distribution Interface provides the BFD base module interface to the BFD Hello module. The Hello module can either be co-located within the process, in a separate process on the same card or another card, or in the hardware. The distribution interface provides a layer of abstraction to the BFD base module so that it can work independently of the system architecture. Typical operations are to set various timer values, and the BFD mode, whether echo or asynchronous, on a router or an interface. It is also sets parameters for multi-hop BFD sessions.

Distribution Line Card Interface

The BFD Distribution Line Card Interface provides the BFD packet processing module from the interface to the BFD Base module. The Packet module can either be co-located within the base module process, in a separate process on the same card or another card, or in the hardware. The Line Card Interface module provides a layer of abstraction to the BFD packet processing module so that it can work independently of the system architecture. Typical operations for this module are to get information from the Base module, and send information back to the Main card module. It is also used to set parameters for multi-hop BFD sessions.

BFD-NSM Interface

The BFD to NSM interface provides an interface to NSM for forwarding plane liveliness detection of static routes. Typical operations for this model are creating a new session, or deleting or updating an existing session. It is also necessary to specify BFD Graceful-Restart parameters for a session.

System Design

BFD can run in a pizza box or on a chassis-based system. For a pizza box, the monolithic version of BFD will generally be used, although the distributed version, with the hello-sending functionality, running either as a separate process or in the hardware, can also be used. For a chassis-based system, it is generally advisable to run the BFD base module, along with other control plane processes, in the central control processor. The BFD Hello processing modules can be distributed across the line cards where the Hello processing is run either in the software or in the hardware.

Software Integration

BFD base module interacts with other modules, such as NSM and protocol modules, via the BFD client. The BFD client library in the protocol modules connects to the BFD server library in the BFD base module using the socket interface.

Session Performance Table

The BFD session performance table (bfdSessPerfTable) is used for collecting BFD performance counts on a per session basis. This table is an augment to the BFD session table; therefore the index for this table is bfdSessIndex. An entry in this table is created by a BFD-enabled node for every BFD Session. The bfdCounterDiscontinuityTime is used to indicate potential discontinuity for all counter objects in this table. All objects in this table are of read-only type, so no Set operations are allowed. The Session Table functions are also used to retrieve the BFD sessions. BFD maintains an array of two 32-bit value to support both 64-bit and 32-bit versions of a counter object. When SNMP requests are received for a 32-bit object, the 0th index of the array is used to return value to the user. When SNMP requests are received for a 64-bit object, both the 0th and the 1st index of the array are returned to the user.

BFD Session Discriminator Mapping Table

The BFD Session Discriminator Mapping Table (bfdSessDiscMapTable) maps a local discriminator value to the associated BFD session's BfdSessIndexTC used in the bfdSessTable. The index used is the BFD session's local discriminator. The object in this table is of type read-only, therefore, no Set operation is allowed.

CHAPTER 3 BFD Application Protocol Modules

This chapter discusses the application protocol modules that interact with the BFD protocol.

Modules

The following subsection describes the external modules that interact with the BFD client. Refer to Chapter 2, *BFD Software Architecture* to see a diagram of this interaction.

BFD NSM Static Route Module

BFD for NSM static routes is a core module that handles the static route functionality based on BFD session updates. NSM searches all Routing Information Base (RIB) entries upon receiving a BFD session event, which uses the specified next-hop address and outgoing interface from its routing table entries. The next-hop could match any one of the multiple next-hops stored in the RIB entry, which may be in either the active or inactive state.

Refer to Chapter 6, *Network Services Module* for more information.

BFD BGP Module

This is the plug-in module for BGP to support BFD capability for both IPv4 and IPv6. It manages the following tasks in addition initialization and termination of the BFD client module:

- BFD session handling for the BGP peer-reachability detection
- Callback registration for the BFD session-event handling

The BGP BFD module applies BFD as the peer-reachability detection for both iBGP peers and eBGP peers. It adds the BFD session when a BGP peer reaches the ESTABLISHED state and deletes it if the peer moves down from the ESTABLISHED state.

Refer to Chapter 7, *Border Gateway Protocol* for more information.

BFD IS-IS Module

This is the plug-in module for IS-IS to support BFD capability for both IPv4 and IPv6. It manages the following tasks in addition to initialization and termination of the BFD client module:

- BFD session handling for IS-IS neighbor-reachability detection
- Callback registration for BFD session-event handling

This module applies BFD as the neighbor-reachability detection protocol for both a shared link and a point-to-point link. It adds a BFD session when an IS-IS neighbor is in the Full state, except for a shared link. A session is added only when either the neighbor or itself is Designated Intermediate System (DIS) on a shared link to prevent an unnecessary full mesh of the BFD session on shared media. It deletes the BFD session when the neighbor is below Full state.

Refer to Chapter 8, *Intermediate System To Intermediate System* for more information.

BFD OSPFv2 Module

This is the plug-in module for OSPFv2 to support BFD capability. It manages the following tasks:

- BFD session handling for OSPF neighbor reachability detection
- Callback registration for BFD session-event handling

The OSPFv2 BFD module applies BFD as the neighbor-reachability detection protocol for both standard shared links and OSPF virtual links. It adds a BFD session when an OSPF neighbor goes beyond the OSPF two-way state, thus avoiding an unnecessary full mesh of the BFD session on shared links.

Refer to Chapter 9, *Open Shortest Path First* for more information.

BFD RIP Module

This is the plug-in module for RIP to support BFD capability. It manages the following tasks:

- BFD session handling for RIP neighbor reachability detection
- Callback registration for BFD session-event handling

When RIP-BFD support is enabled for all interfaces or a specific neighbor, a new BFD session is added. RIP registers with BFD, which triggers the creation of a BFD session for a neighbor. BFD then creates a session for the given neighbor.

RIP maintains a configured neighbor list in a RIP instance for all neighbors for which BFD is enabled. New neighbors are automatically enabled for BFD when the update packets are received.

Refer to Chapter 10, *Routing Information Protocol* for more information.

Interfaces

The following subsection describes the external interfaces that interact with the BFD client.

Application Protocol Interface

The BFD Application Protocol Interface is the interface to the application protocols requiring forwarding plane liveliness and nexthop data plane failure detection. Typical operations supported are to create, delete or update sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD NSM Management Interface

The BFD NSM interface provides forwarding plane liveliness and nexthop data plane failure detection of static routes. Typically, this module creates, deletes or updates sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD OSPFv2 Management Interface

The OSPFv2 management interface provides forwarding plane liveliness detection of static routes. Typically, this module creates, deletes or updates sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD IS-IS Management Interface

The BFD IS-IS interface provides forwarding plane liveliness detection of static routes. Typically, this module creates, deletes or updates sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD BGP Management Interface

The BFD BGP interface provides forwarding plane liveliness detection of static routes. Typically, this module creates, deletes or updates sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD RIP Management Interface

The BFD RIP management interface provides forwarding plane liveliness detection of RIP neighbors. Typically, this module creates, deletes or updates sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD NSM Static Interface

The BFD Static interface detects the static route nexthop data-plane failure. Typically, this module creates, deletes or updates sessions.

CHAPTER 4 Data Structures

This chapter describes the BFD data structures.

Note: The `nsm_master` data structure is common for all ZebOS-XP protocols and is used in BFD functions. See the *Common Data Structures Developer Guide* for a description of this data structure.

bfd_master

This data structure contains the system wide configuration parameters and variables that are used with the BFD Master. It is defined in `oamd/oam.h`.

Member	Description
vr	Virtual router (of type ipi_vr)
nc	NSM client (of type nsm_client)
zg	Library globals (of type lib_globals)
bfd	BFD instance (of type list)
flags	Flags
if_table	Interface table (of type avl_tree)
mh_table	Multihop table (of type avl_tree)
notifiers	Notifier events (of type list)
traps	Traps of type vector
image_type	Bfd image type whose value are: • BFD_IMAGE_MONOLITHIC • BFD_IMAGE_DIST_MAIN • BFD_IMAGE_DIST_LINE
debug	Debugging flags one per virtual router
cfg_debug	Debugging flags one per virtual router
bfd_notify_flag	BFD Notification flag
mpls_oam_list	OAM master data list (of type list)
oam_recvd_req_list	OAM received request list (of type list)
oam_vc_cache_list	OAM Virtual Circuit (VC) cache list (of type list)
mpls_oam_read	OAM read thread
oam_sock	OAM socket descriptor

Member	Description
oam_s_sock	OAM socket descriptor
oam_ipv4_nhop_cache	OAM nexthop cache (of type route_table)

Definition

struct bfd_master

```
{
  /* Pointer to VR. */
  struct ipi_vr *vr;
  /* NSM Client. */
  struct nsm_client *nc;
  /* Pointer to globals.  */
  struct lib_globals *zg;
  /* BFD instance list. */
  struct list *bfd;
  /* BFD global flags */
  u_char flags;
  /* Tree of all interfaces in the system */
  struct avl_tree *if_table;
  /* Tree of all multihop session parameters in the system */
  struct avl_tree *mh_table;
  /* BFD notifiers. */
struct list *notifiers [BFD_NOTIFY_MAX];
  /* BFD SNMP trap callback function */
#ifdef HAVE_SNMP
  vector traps [BFD_TRAP_ID_MAX];
#endif /* HAVE_SNMP */
  /* BFD image type - Monolithic or distributed */
  u_char image_type;
#define BFD_IMAGE_MONOLITHIC     1
#define BFD_IMAGE_DIST_MAIN      2
#define BFD_IMAGE_DIST_LINE      3
  /* Debugging flags one per VR */
  u_int32_t debug;
  u_int32_t cfg_debug;
  /* BFD Notification flag */
  bool_t bfd_notify_flag;
#ifdef HAVE_MPLS_OAM
  /* MPLS OAM Master Data */
  struct list *mpls_oam_list;
  struct list *oam_recvd_req_list;
  struct list *oam_vc_cache_list;
  struct thread *mpls_oam_read;
  int oam_sock;
  int oam_s_sock;
  struct route_table *oam_ipv4_nhop_cache;
#endif /* HAVE_MPLS_OAM */
```

```
};
```

bfd

This data structure contains the configuration parameters and structures that are used with BFD. It is defined in the `oamd/oam.h` file.

Member	Description
bfd_id	BFD identifier
next_sock	Next socket descriptor
start_time	BFD start time
bm	BFD master (of type bfd_master)
bv	BFD VRF binding (of type bfd_vrf)
router_id	Router Identifier address (of type pal_in4_addr)
flags	Administrative flag whose values are the following: • BFD_PROC_UP • BFD_PROC_DESTROY
fd	BFD socket descriptor for read and write operations
config	Configuration variable
proto_info	Protocol info (of type bfd_proto_info)
bi	BFD global interface structure
Sess	List of all session for particular instance (of type list)
loc_disc	Local discriminator
g_echo_allowed	Global echo mode
g_slow_timer	Globally configured slow timer value
trap_count	Trap count
notify_time	Notification time (of type pal_time_t)
sess_key_main	Session key main (of type avl_tree)
sess_mpls	Session table based on FTN ID related to MPLS (of type avl_tree)
sess_mpls_egress	Session table based on Remote Disc, Destination address
sess_vccv	Session table based on incoming VC label
sess_fwd_vccv	Session table based on incoming VC label for Virtual Circuit Connection Verification (VCCV) related session
sess_index	Session table based on index

Member	Description
t_read_async	Thread for async (of type thread)
t_read_mhop	Thread for multihop operation (of type thread)
t_read_echo	Thread for echo operation (of type thread)
sock_async	Socket descriptor for asynchronous operation
sock_echo	Socket descriptor for echo operation
sock_mhop	Socket descriptor for multihop operation
sock_echo_send	Raw socket descriptor
sock_async_send	UPD socket descriptor
t_read_async6	Thread for async of IPV6 (of type thread)
t_read_echo6	Thread for echo operation for IPV6 (of type thread)
t_read_mhop6	Thread for multihop operation for IPV6 (of type thread)
sock_echo6_send	Socket descriptor for echo operation of IPV6
sock_async6_send	Socket descriptor for asynchronous operation of IPV6
sock_multihop6_send	Socket descriptor for multihop operation of IVP6
sess_key	BFD session structure tree (of type avl_tree)
sess_disc	Session tree based on discriminator
obuf	Output buffer
rcv_pkt	Session pre-allocated receive packet

Definition

```
struct bfd
{
  /* BFD ID. */
  u_int16_t bfd_id;
  u_int16_t next_sock;
  /* BFD start time. */
  pal_time_t start_time;
  /* Pointer to BFD master. */
  struct bfd_master *bm;
  /* BFD VRF binding */
  struct bfd_vrf *bv;
  /* BFD Router ID. */
  struct pal_in4_addr router_id;                /* BFD Router-ID. */
  /* Administrative flags. */
  u_int16_t flags;
#define BFD_PROC_UP                   (1 << 0)
```

```
#define BFD_PROC_DESTROY               (1 << 1)
  /* BFD socket for read/write. */
  s_int32_t fd;

  /* Configuration variables. */
  u_int16_t config;
  struct bfd_proto_info proto_info [IPI_PROTO_MAX];
  /* Global interface structure - For all information not part of any
     particular interface*/
  struct bfd_interface *bi;
#ifdef HAVE_BFD
  /* List of all session for this instance */
  struct list *sess;
#if defined (HAVE_BFD_MONO) || defined (HAVE_BFD_MAIN)
  /* Next local discriminator to be given */
  u_int32_t loc_disc;
  /* Global echo mode */
  bool_t g_echo_allowed;
  /* Globally configured slow timer value */
  u_int32_t g_slow_timer;
  /* Trap variables */
  u_int16_t trap_count;
  pal_time_t notify_time;
  /* Configure tables information from applications - bfd_session structure tree */
  struct avl_tree *sess_key_main [BFD_ADDR_FAMILY_MAX];
#ifdef HAVE_MPLS_OAM
  /* Session table based on FTN ID. This is for MPLS related sessions at Ingress -
bfd_session structure tree. */
  struct avl_tree *sess_mpls;
  /* Session table based on Remote Disc, Destination Address. This is for MPLS related
sessions at Egress - bfd_session structure tree. */
  struct avl_tree *sess_mpls_egress;
#endif /* HAVE_MPLS_OAM */
#ifdef HAVE_VCCV
  /* Session table based on Incoming VC Label. This is for  VCCV related sessions at
Ingress - bfd_session structure tree.
   */
  struct avl_tree *sess_vccv;
  /* Session table based on Incoming VC Label. This is for VCCV related sessions'
forwarding entries - bfd_session_fwd
   structure tree. */
  struct avl_tree *sess_fwd_vccv;
#endif /* HAVE_VCCV */
  /* Session table based on index  - use the my_discriminator */
  struct avl_tree *sess_index;
#endif /* HAVE_BFD_MONO || HAVE_BFD_MAIN */
#if defined (HAVE_BFD_MONO) || defined (HAVE_BFD_LINE)
  /* Structure sockets for the monolithic or line card case - not required for
     distributed main card case */
  struct thread *t_read_async;
  struct thread *t_read_mhop;
```

```
  struct thread *t_read_echo;
  /* udp recv socket for IPv4 */
  s_int32_t sock_async;
  s_int32_t sock_echo;
  s_int32_t sock_multihop;
  s_int32_t sock_echo_send; /* Raw socket */
  s_int32_t sock_async_send; /* UDP socket */
#ifdef HAVE_IPV6
  struct thread *t_read_async6;
  struct thread *t_read_echo6;
  struct thread *t_read_mhop6;
  /* udp recv socket for IPv6 */
  s_int32_t sock_async6;
  s_int32_t sock_echo6;
  s_int32_t sock_multihop6;
  s_int32_t sock_echo6_send; /* Raw socket */
  s_int32_t sock_async6_send; /* UDP socket */
  s_int32_t sock_multihop6_send;
#endif /* HAVE_IPV6 */
  /* Session based on IP Address/ ifindex from applications - bfd_session_fwd  structure
tree  */
  struct avl_tree *sess_key [BFD_ADDR_FAMILY_MAX];
  /* Session tree based on your discriminator - bfd_session_fwd structure tree   */
  struct avl_tree *sess_disc;
  /* Output buffer and stream */
  struct stream *obuf;
  /* Session pre-allocated receive packet */
  u_char rcv_pkt [1600] ;
#endif  /* HAVE_BFD_MONO || HAVE_BFD_LINE */
#endif /* HAVE_BFD */
};
```

rip_master

This data structure contains the system wide configuration parameters and variables that are used with the RIP Master. It is defined in the `ripd/ripd.h` file.

Member	Description
vr	Pointer to VR
zg	Pointer to globals
rip	RIP instance list
config	RIP global configuration
flags	RIP global flags
if_table	RIP global interface table
if_params	RIP interface parameter pool

Member	Description
conf	Debug flags for configuration
term	Debug flags for terminal
global_route_changes	RIP route changes
global_queries	RIP queries
grace_period	RIP grace period

Definition

```
struct rip_master
{
  /* Pointer to VR.  */
  struct ipi_vr *vr;

  /* Pointer to globals.  */
  struct lib_globals *zg;

  /* RIP instance list.  */
  struct list *rip;

  /* RIP global configuration.  */
  u_char config;
#define RIP_GLOBAL_CONFIG_RESTART_GRACE_PERIOD  (1 << 0)

  /* RIP global flags.  */
  u_char flags;
#define RIP_GRACEFUL_RESTART                    (1 << 0)

  /* RIP global interface table.  */
  struct route_table *if_table;

  /* RIP interface parameter pool.  */
  struct list *if_params;

  /* RIP debug flags.  */
  struct
  {
    /* Debug flags for configuration.  */
    struct debug_rip conf;

    /* Debug flags for terminal.  */
    struct debug_rip term;

  } debug;

  /* RIP route changes.  */
  int global_route_changes;
```

```
  /* RIP queries.  */
  int global_queries;

#ifdef HAVE_RESTART
  /* RIP grace period.  */
  u_int32_t grace_period;
#endif /* HAVE_RESTART */
};
```

bfd_client_session_info

This data structure defines the configuration parameters that are used with the BFD client session. It is defined in the `oamd/bfd/Bfd_common.h` file.

Member	Description
client	Client identifier
ifindex	Interface index
cli_ifindex	BFD session interface index
flags	Flag whose values are: • BFD_MSG_SESSION_FLAG_MH • BFD_MSG_SESSION_FLAG_DC • BFD_MSG_SESSION_FLAG_PS • BFD_MSG_SESSION_FLAG_AD
ll_type	Prefix length
sess_type	Session type
src_addr	Source address (of type prefix)
dst_addr	Destination address (of type prefix)
min_tx	BFD minimum transmission interval
min_rx	BFD minimum reception interval
multiplier	BFD detection multiplier
rem_disc	Remote descriminator
mpls_params	Parameters for sessions related to MPLS LSP FEC
vccv_params	Parameters for VCCV related sessions
sess_index	Session index

Definition

```
struct bfd_client_session_info
{
```

```
  /* Client id  */
  module_id_t client;

  /* Interface index.  */
  u_int32_t ifindex;

  /* bfd session's interface index */
  u_int32_t cli_ifindex;

  /* Flags of the session.  */
  u_char flags;
#define BFD_MSG_SESSION_FLAG_MH    (1 << 0)  /* Multi-Hop.  */
#define BFD_MSG_SESSION_FLAG_DC    (1 << 1)  /* Demand Circuit.  */
#define BFD_MSG_SESSION_FLAG_PS    (1 << 2)  /* Persistent Session.  */
#define BFD_MSG_SESSION_FLAG_AD    (1 << 3)  /* User Admin Down.  */

  /* Prefix length.  */
  u_char ll_type;

  /* Session type - not session type cannot be made just a flag */
  u_char sess_type;

  /* Source and destination addresses.  */
  struct prefix src_addr;
  struct prefix dst_addr;

  /* BFD min Tx interval received from Application. */
u_int32_t min_tx;

  /* BFD min Rx interval recieved from Application. */
  u_int32_t min_rx;

  /* BFD Detection Multiplier recieved from Application. */
  u_int32_t multiplier;

  /* BFD Remote Discriminator received through BFD TLV in
   * LSP Ping for MPLS sessions at Egress.
   */
  u_int32_t rem_disc;

#ifdef HAVE_MPLS_OAM
  union
  {
    /* BFD Parameters for MPLS LSP FEC related sessions. */
    struct bfd_mpls_params mpls_params;
#ifdef HAVE_VCCV
    /* BFD Parameters for VCCV related sessions.  */
    struct bfd_vccv_params vccv_params;
#endif /* HAVE_VCCV */
  }addl;
```

```
#endif /* HAVE_MPLS_OAM */

#ifdef HAVE_SNMP
  u_int32_t sess_index;
#endif /*HAVE_SNMP */
};
```

bfd_event_session

The bfd_event_session enumeration defines all the session events.

Constant	Description
BFD_EVENT_SESSION_ADMIN_DOWN	The session is administratively configured to down
BFD_EVENT_SESSION_DOWN	Initial state on session creation
BFD_EVENT_SESSION_INIT	Transits to “init” on reception of Down message from remote peer
BFD_EVENT_SESSION_UP	Transits to “UP” state on successful completion of a three-way handshake
BFD_EVENT_SESSION_MAX	Session Maximum

Definition

```
enum bfd_event_session
{
  BFD_EVENT_SESSION_ADMIN_DOWN = 0,
  BFD_EVENT_SESSION_DOWN,
  BFD_EVENT_SESSION_INIT,
  BFD_EVENT_SESSION_UP,
  BFD_EVENT_SESSION_MAX
};
```

bfd_notify_event

The bfd_notify_event enumeration defines all the BFD notification events.

Constant	Description
BFD_NOTIFY_PROCESS_NEW	When a new process is added
BFD_NOTIFY_PROCESS_DEL	When a process is deleted
BFD_NOTIFY_ROUTER_ID_CHANGED	When router identifier is changed
BFD_NOTIFY_LINK_NEW	When a new link is added
BFD_NOTIFY_LINK_DEL	When a link is deleted
BFD_NOTIFY_LINK_UP	When a link is up
BFD_NOTIFY_LINK_DOWN	When a link is down
BFD_NOTIFY_ADDRESS_NEW	When a new address is added

Constant	Description
BFD_NOTIFY_ADDRESS_DEL	When an address is deleted
BFD_NOTIFY_MAX	Maximum

Definition

```
enum bfd_notify_event
{
  BFD_NOTIFY_PROCESS_NEW = 0,
  BFD_NOTIFY_PROCESS_DEL,
  BFD_NOTIFY_ROUTER_ID_CHANGED,
  BFD_NOTIFY_LINK_NEW,
  BFD_NOTIFY_LINK_DEL,
  BFD_NOTIFY_LINK_UP,
  BFD_NOTIFY_LINK_DOWN,
  BFD_NOTIFY_ADDRESS_NEW,
  BFD_NOTIFY_ADDRESS_DEL,
  BFD_NOTIFY_MAX
};
```

bfd_storage_type

The bfd_storage_type enumeration defines all the BFD storage types.

Constant	Description
ST_OTHER	When storage is other
ST_VOLATILE	When storage type is volatile
ST_NONVOLATILE	When storage type is nonvolatile
ST_PERMANENT	When storage is permanent
ST_READONLY	When storage is read only

Definition

```
enum bfd_storage_type
{
  ST_OTHER = 1,
  ST_VOLATILE,
  ST_NONVOLATILE,
  ST_PERMANENT,
  ST_READONLY
};
```

CHAPTER 5 OAM for BFD and MPLS

This chapter describes the ZebOS-XP Operations, Administration, and Maintenance Daemon (`oamd`), which manages both the BFD and Multiprotocol Label Switching (MPLS) OAM. Base BFD functionality resides within the OAM module. Included in this chapter are descriptions of the OAM command functions.

Overview

OAM for MPLS and BFD manages the following:

- Processing ping or trace requests from MPLS ONM
- Sending requests and processing replies to or from Network Service Module (NSM) to fetch label information and label switched paths (LSP) Target Forwarding Equivalency Class (FEC) stack information to encode or decode LSP Echo requests or reply packets
- Providing the ability to trigger periodic LSP ping packets from modules within OAMD
- Reusing the inputs received from NSM for further requests or reply messages if the node is the Ingress or Egress node for the FEC in the message
- Establishing communication with the MPLS Forwarder (`mplsfwd`) module to send and receive MPLS OAM packets to or from the forwarder
- Sending periodic LSP Ping Echo requests with additional BFD TLV for an FEC when a BFD session is configured for that FEC at Ingress
- Creating a BFD session when a valid LSP ping echo request with a BFD TLV is received at Egress
- Bringing down the BFD session when an LSP ping failure occurs

BFD Support for MPLS

To support MPLS LSP, the BFD module has been enhanced to handle the following capabilities:

- Creating new trees to store MPLS LSP-related sessions at Ingress and Egress nodes
- De-multiplexing received BFD control packets related to MPLS sessions using a remote discriminator (RD) field
- Sending BFD control packets related to MPLS sessions to the MPLS forwarder to send the packets over MPLS LSP with necessary label encapsulation

BFD Support for VCCV

To support BFD Virtual Circuit Connectivity Verification (VCCV) sessions, BFD has been enhanced to handle the following capabilities:

- Creating new trees for BFD-VCCV sessions and session forwarding entries
- Sending BFD control packets related to VCCV to the MPLS Forwarder to send the packets over LSP with the necessary label encapsulation

Architecture

The architecture of the MPLS OAM feature is depicted in Figure 5-1, below.

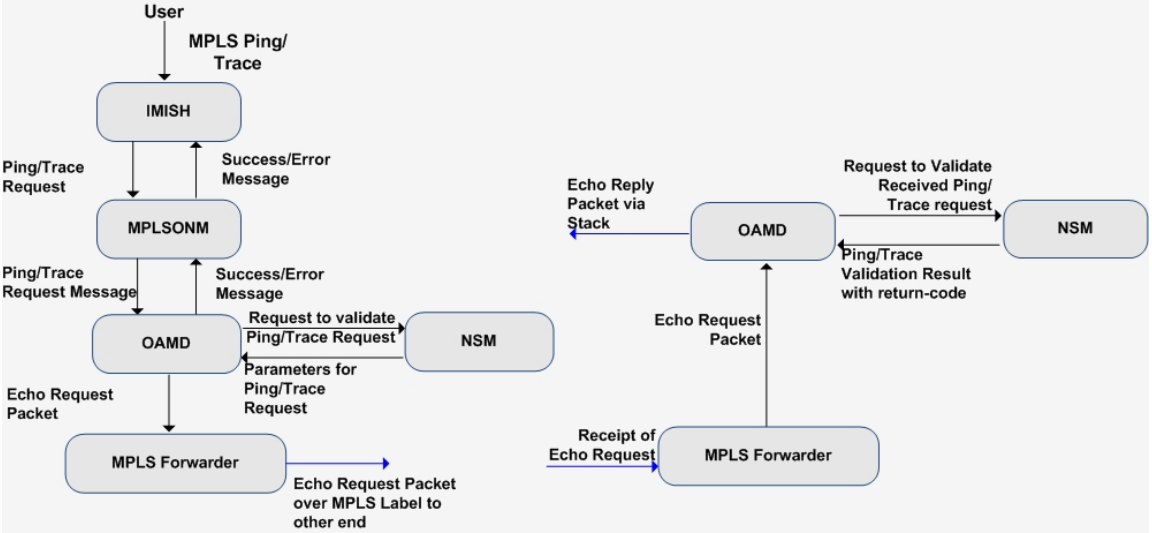

Figure 5-1: Module interaction for MPLS OAM

Virtual Circuit Connectivity Verification

Virtual Circuit Connectivity Verification provides a control channel (CC) between a pseudowire's ingress and egress points, over which connectivity verification (CV) messages can be sent. Connectivity messages are used for end-to-end fault detection and diagnosis to determine a pseudowire's true operational state.

The design of VCCV comprises the following capabilities:

- A means of signaling BFD-VCCV capabilities of a local PE (provider edge) to a remote PE
- Encapsulation of the BFD-VCCV control channel messages that allow the receiving PE to intercept, interpret, and process them locally as OAM messages
- Provision for the operation of various VCCV operational modes transmitted within the VCCV messages

Signaling Capabilities from Local PE to Remote PE

Dynamic Virtual Circuits

When a pseudowire (PW) is initially signaled using the Label Discovery Protocol (LDP), a label mapping message is sent from the initiating PE to the receiving PE requesting that a pseudowire be set up. The label mapping message has been extended to include BFD-VCCV capability information. This information informs the receiving PE about the combination of Control Channel (CC) and BFD Connectivity Verification (CV) types, and whether the sending PE is capable of receiving PE. If the receiving PE agrees to establish the PW, it returns its capabilities in the subsequent signaling message to indicate the CC and (BFD) CV types it is capable of processing.

Static Virtual Circuits

During Virtual Circuit (VC) creation, if VCCV requires a VC, the user has to specify the CC type to use. If BFD-VCCV is required, then the BFD CV type must be provided as input.

Note: It is not necessary to specify a CV type during VC creation.

Control Channel Messages

VCCV encapsulation allows the control channel to be processed similar to data traffic for the PW, in order to test the data plane at the PE. It also allows the PE to intercept and process VCCV messages instead of forwarding them out of the Access Circuit (AC) toward the Customer Edge (CE) as if they were data traffic. For MPLS PWs, the following CC types and CV types are supported. Based on the CC type and CV type, a VCCV message is encapsulated.

Control Channel Types

- Type 1: PWE3 Control Word with 0001b as first nibble
- Type 2: MPLS Router Alert Label
- Type 3: MPLS PW Label with TTL == 1

Connectivity Verification Types

- LSP Ping
- BFD IP/UDP-encapsulated, for PW Fault Detection only
- BFD IP/UDP-encapsulated, for PW Fault Detection and AC/PW Fault Status Signaling
- BFD PW-ACH-encapsulated, for PW Fault Detection only
- BFD PW-ACH-encapsulated, for PW Fault Detection and AC/PW Fault Status Signaling

Operational Modes

VCCV Control Channel type defines the control channels that VCCV can support. VCCV supports multiple CV types concurrently, but it only supports the use of a single CC type. A VCCV Control Channel can be in-band, and follow the same path as PW data, or it can be out-of-band, meaning that the path may be different, because of the Equal Cost Multi-path (ECMP) behavior applied at the nodes.

BFD VCCV Capabilities Signaling

Figure 5-2 depicts BFD VCCV capabilities signaling.

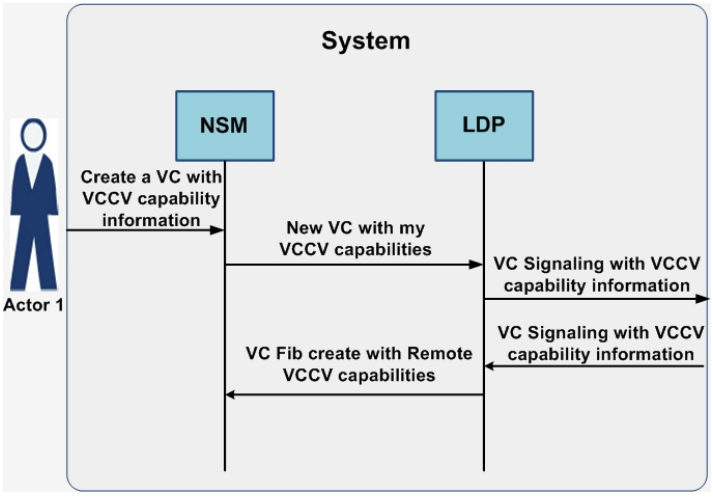

Figure 5-2: BFD VCCV Capabilities Signaling

VCCV LSP Ping

Figure 5-3 depicts the sequence of operations for a VCCV LSP ping request.

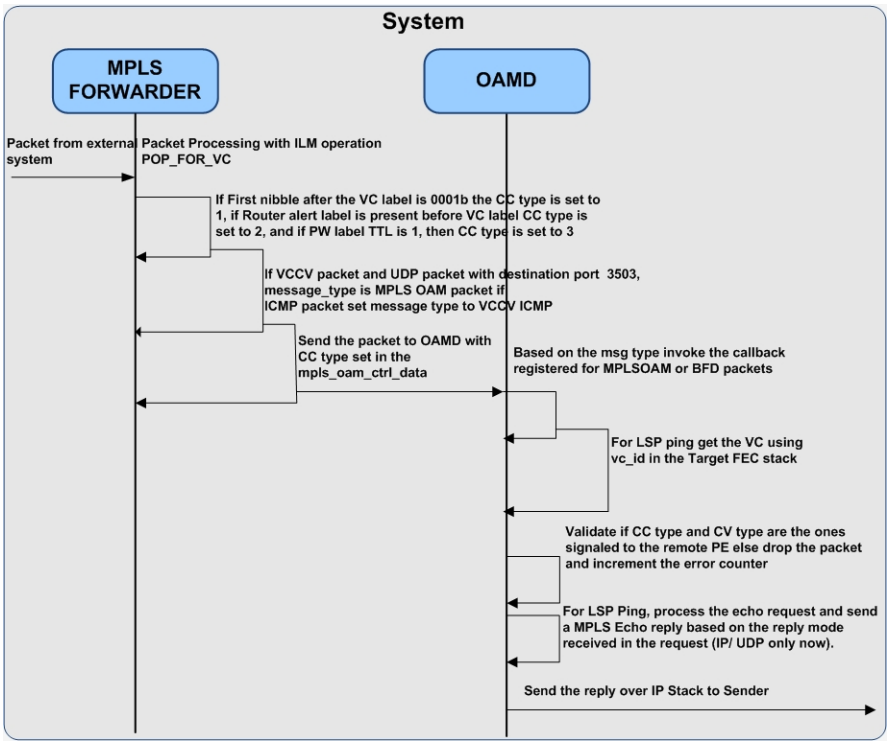

Figure 5-3: Sequence Diagram for VCCV LSP Ping

VCCV LSP Ping Request Processing

Figure 5-4 depicts the sequence for processing a VCCV LSP ping request.

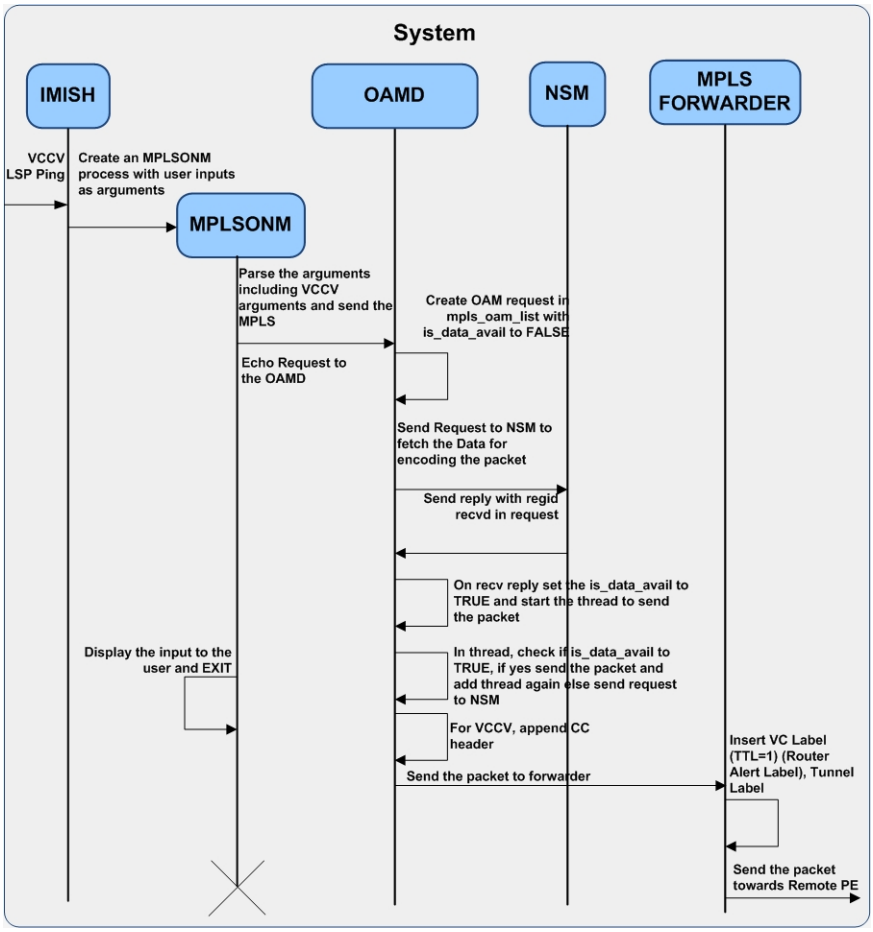

Figure 5-4: Sequence Diagram for VCCV LSP Ping Request Processing

BFD for MPLS LSP and VC

BFD is used to detect an MPLS LSP data plane failure and periodic verification of VC using VCCV control channels.

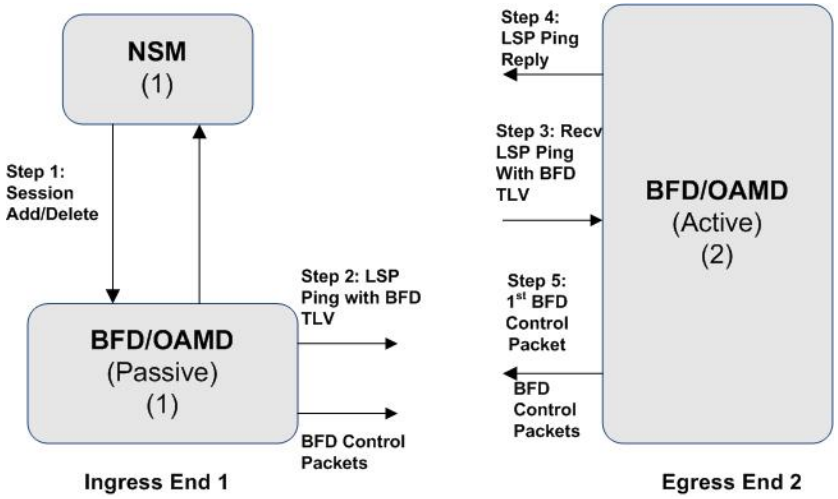

Figure 5-5: Data Flow Diagram for BFD-MPLS

At Ingress—End 1

In Figure 5-5, steps 1 through 5 are executed in the same order, then BFD and LSP ping run independently.

Step 1. NSM MPLS asks BFD for session addition or deletion information for an FEC with `ftn-index` as identifier and link-layer type as `bfd_ll_mpls_lsp`.

Step 2. BFD, upon receiving a session add message from NSM with a link-layer type of `bfd_ll_mpls_lsp`, generates a My Discriminator (MD) for the session, and adds an LSP ping echo request in the global `oam_req_list` with `bfd_req` flag set.

The OAMD module sends an LSP ping echo request message towards the Egress with the BFD TLV set, including the local discriminator (LD) value. At this point, BFD is in passive state.

At Egress—End 2

Step 3. Upon receipt of an LSP ping echo request with a BFD TLV, the FEC in the request is validated, then a multi-hop BFD session is added with the MD of the the ingress LSR (label switched route) as the remote discriminator and link-layer type as `bfd_ll_mpls_lsp`.

Step 4. If the LSP ping echo request message is valid, an echo reply is sent to the ingress node. The echo reply message may have the BFD TLV. For an initial echo reply packet, the BFD TLV is not inserted, but the local discriminator for that session is generated by BFD and included in the packet.

Step 5. At Egress, BFD runs in the active state, sending out the first BFD control packet with remote discriminator filled with the value received in the LSP ping echo request message.

BFD for VCCV

Figure 5-6 depicts the data flow for BFD for VCCV.

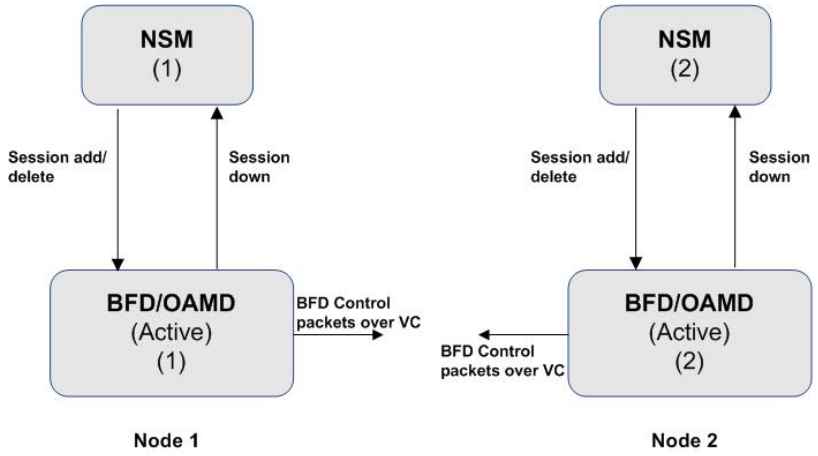

Figure 5-6: Data Flow Diagram for BFD VCCV

- At Ingress, an LSP ping echo request message is periodically sent at an interval configured by the user, while configuring BFD for the FEC. These periodic LSP ping echo requests messages always include the BFD TLV.
- The BFD session at the Ingress node takes the passive role and waits for the BFD messages to be received from the Egress node before sending BFD control packets.
- BFD control packets are sent to the MPLS Forwarder for encapsulation with an MPLS LABEL, and then forwarded over the LSP for that FEC.

- When an LSP ping fails (five consecutive echo time-outs occur) and the corresponding BFD session for that FEC is UP, then the BFD session is brought down. A session Down message is sent to NSM, which in turn notifies the LSP owner about its state.
- NSM sends a BFD session delete message to BFD for an FEC when the FEC is removed or when the BFD configuration for the FEC is removed. In the latter case, the ADMIN_DOWN flag is set in the session-delete message.
- At Egress, BFD session is boot-strapped with the LSP Ping with the BFD TLV set. For sessions at the Egress, the BFD module itself acts as the protocol client.
- BFD takes the active role at Egress. The My Discriminator information received from the Ingress LSR is used as the remote discriminator of the BFD session. As part of FSM processing, control packets are then routed to the Ingress LSR directly through the stack.
- At the Egress, after a session is created (in FSM Down state), a timer is added with a hold time of five seconds. If no more packets come from the Ingress for this session within the hold time, the session is deleted.
- When a local discriminator is generated for a session at Egress, and is updated to the OAM module in BFD, the OAM module does not send any further session-add messages to BFD upon receipt of subsequent echo requests with BFD TLVs for the same FEC from the same source.
- At Egress, when a BFD session goes down, or if an Admin Down notificationis received from Ingress, a timer is activated with a timeout value calculated as “Detect. Interval * Negotiated Tx interval”. If the session does not come UP before the timer expires, it is deleted.
- All BFD sessions for an LSP (from Ingress to Egress) are single-hop sessions; however, in the reverse direction (from Egress to Ingress), they are multi-hop sessions.
- The destination UDP port used for ingress-to-egress BFD single-hop sessions is 3784. The destination UDP port used for egress-to-ingress BFD multi-hop sessions is 4784.
- BFD control packets and LSP-Ping packets have the same destination address prefix in the IP header. This address is in the 127/8 address range.
- NSM sends a BFD session add message to BFD module when VC gets installed in the Forwarder and BFD is configured for that VC. The link layer type for VCCV related sessions is set to ‘bfd_ll_mpls_vccv’.
- Additional parameters sent to the BFD module to enable BFD to encapsulate and send BFD control packets over VC are VC ID, tunnel label, outgoing VC label, CC type, CV type, access interface index (ac_ix).
- On receiving BFD session-add message from NSM, BFD creates a single-hop session in the `sess_vccv` AVL tree with incoming VC Label as the key. The Label information and the CC type, CV type received during session add are stored in the `bfd_vccv_params` structure in the `bfd_session` and the `bfd_session_fwd` structures.
- A session-add request is also sent to the BFD packet processing layer. In this layer, a new session forward entry is created and added to the `sess_fwd_vccv` tree. The key to the tree is the incoming VC Label. The tree is used to identify the VCCV session for the first BFD control packet received with 0 in the remote discriminator field.
- Once the session is added, based on the CV type, a BFD control packet is constructed and sent to the MPLS forwarder based on the encapsulation type. The following encapsulation types are defined for BFD VCCV packets:
  - If CV type is 0x20 or 0x10, BFD PW-ACH-encapsulation is used (without IP/UDP headers)
  - If CV type is 0x08 or 0x04, BFD IP/UDP-encapsulation is used
- BFD runs in active mode at both ends for VCCV-related sessions.
- When a BFD control packet is received for VCCV-related sessions from the MPLS Forwarder, packet sanity is verified first. If a valid session exits, then the CC type over which the packet was received and the CV type of the packet is verified with the CC type and CV type for that VC. If they do not match. the packet is dropped, and the global counter `ivccv_discards` is incremented.
- When a VC is removed, NSM sends a BFD session-delete message for the VC and the VC FIB is deleted.
- A BFD session for a VC can be administratively brought down using a `ping mpls` command invoked by the user.

Command API

The functions defined in this chapter are called by the OAM commands defined in the *NSM Command Reference*.

Function	Application
nsm_mpls_bfd_api_fec_set	Sets the BFD fall-over check for the FEC of an LSP type (LDP, Resource Reservation Protocol (RSVP) or Static)
nsm_mpls_bfd_api_fec_unset	Removes the BFD fall-over check for the FECs of an LSP type
nsm_mpls_bfd_api_fec_disable_set	Disables the BFD fall-over check for the FEC or Trunk of an LSP type
nsm_mpls_bfd_api_lsp_all_set	Sets the BFD fall-over check for all FECs of an LSP type
nsm_mpls_bfd_api_lsp_all_unset	Removes the BFD fall-over check for all FECs of an LSP type
nsm_mpls_bfd_api_vccv_trigger	Starts or stops the BFD VCCV session for a VC based on the specified operation

nsm_mpls_bfd_api_fec_set

This function sets the BFD fall-over check for the FEC of an LSP-type (LDP, RSVP, or Static).

Syntax

```
int
nsm_mpls_bfd_api_fec_set (u_int32_t vr_id, char **lsp_name, char *input,
                          u_int32_t lsp_ping_intvl, u_int32_t min_tx,
                          u_int32_t min_rx, u_int32_t mult,
                          bool_t force_explicit_null)
```

Input Parameters

`vr_id` Virtual router ID of the VR in this context

`lsp_name` LSP type name as LDP, RSVP, or Static

`input` FEC as Prefix or Trunk name

`lsp_ping_intvl`
The LSP Ping Interval in seconds

`min_tx` Minimum BFD transmission in milliseconds

`min_rx` Minimum BFD reception in milliseconds

`mult` BFD Detection Multiplier value

`force_explicit_null`
Flag option

Output Parameters

None

Return Values

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_FAILURE when the nmpls element of the NSM master does not exist

NSM_MPLS_BFD_ERR_LSP_UNKNOWN when the LSP type is unknown (other than LDP, RSVP, or Static)

NSM_MPLS_BFD_ERR_INVALID_INTVL when the lsp_ping_interval is not within the valid range

NSM_MPLS_BFD_ERR_INVALID_PREFIX when the value of input cannot be converted to a valid prefix

NSM_MPLS_BFD_ERR_ENTRY_EXISTS when the BFD entry already exists

NSM_ERR_MEM_ALLOC_FAILURE when a memory allocation error occurs

NSM_SUCCESS when the function succeeds

nsm_mpls_bfd_api_fec_unset

This function removes the BFD fall-over check for the FECs of an LSP type (LDP, RSVP, or Static).

Syntax

```
int
nsm_mpls_bfd_api_fec_unset (u_int32_t vr_id, char **lsp_name,  char *input)
```

Input Parameters

`vr_id`	Virtual router ID of the VR in this context
`lsp_name`	LSP type name as LDP, RSVP, or Static
`input`	FEC as Prefix or Trunk name

Output Parameters

None

Return Values

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_FAILURE when a generic error occurs

NSM_MPLS_BFD_ERR_LSP_UNKNOWN when the LSP type is unknown (other than LDP, RSVP, or Static)

NSM_MPLS_BFD_ERR_ENTRY_NOT_FOUND when the BFD entry does not exists

NSM_SUCCESS when the function succeeds

nsm_mpls_bfd_api_fec_disable_set

This function disables the BFD fall-over check for the FEC or Trunk of an LSP-type.

Syntax

```
int
nsm_mpls_bfd_api_fec_disable_set (u_int32_t vr_id, char *lsp_name, char *input)
```

Input Parameters

`vr_id`	Virtual router ID of the VR in this context
`lsp_name`	LSP type name as LDP, RSVP, or Static
`input`	FEC as Prefix or Trunk name

Output Parameters

None

Return Values

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_FAILURE when the nmpls element of the NSM master does not exist

NSM_MPLS_BFD_ERR_LSP_UNKNOWN when the LSP type is unknown (other than LDP, RSVP, or Static)

NSM_MPLS_BFD_ERR_INVALID_PREFIX when the value of input cannot be converted to a valid prefix

NSM_MPLS_BFD_ERR_ENTRY_EXISTS when the BFD entry already exists

NSM_ERR_MEM_ALLOC_FAILURE when a memory allocation error occurs

NSM_SUCCESS when the function succeeds

nsm_mpls_bfd_api_lsp_all_set

This function sets the BFD fall-over check for all FECs of an LSP type.

Syntax

```
int
nsm_mpls_bfd_api_lsp_all_set (u_int32_t vr_id, char *lsp_name,
                              u_int32_t lsp_ping_intvl, u_int32_t min_tx,
                              u_int32_t min_rx, u_int32_t mult,
                              bool_t force_explicit_null)
```

Input Parameters

`vr_id`	Virtual router ID of the VR in this context
`lsp_name`	LSP type name as LDP, RSVP, or Static
`lsp_ping_intvl`	Specify LSP Ping Interval in seconds
`min_tx`	Minimum BFD transmission in milliseconds
`min_rx`	Minimum BFD reception in milliseconds
`mult`	BFD Detection Multiplier value
`force_explicit_null`	Flag option

Output Parameters

None

Return Values

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_FAILURE when the nmpls element of the NSM master does not exist

NSM_MPLS_BFD_ERR_LSP_UNKNOWN when the LSP type is unknown (other than LDP, RSVP, or Static)

NSM_MPLS_BFD_ERR_INVALID_INTVL when the lsp_ping_interval is not within the valid range

NSM_MPLS_BFD_ERR_INVALID_PREFIX when the FEC prefix/Tunnel-name is invalid

NSM_MPLS_BFD_ERR_ENTRY_EXISTS when the BFD entry already exists

NSM_SUCCESS when the function succeeds

nsm_mpls_bfd_api_lsp_all_unset

This function removes the BFD fall-over check for all FECs of an LSP type.

Syntax

```
int
nsm_mpls_bfd_api_lsp_all_unset (u_int32_t vr_id, char *lsp_name)
```

Input Parameters

`vr_id`	Virtual router ID of the VR in this context
`lsp_name`	LSP type name as LDP, RSVP, or Static

Output Parameters

None

Return Values

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_FAILURE when a generic error occurs

NSM_MPLS_BFD_ERR_LSP_UNKNOWN when the LSP type is unknown (other than LDP, RSVP, or Static)

NSM_SUCCESS when the function succeeds

nsm_mpls_bfd_api_vccv_trigger

This function starts or stops the BFD VCCV session for a VC based on the specified operation.

Syntax

```
int
nsm_mpls_bfd_api_vccv_trigger (struct nsm_master *nm, uint32_t vc_id, u_char op)
```

Input Parameters

`nm`	A pointer to NSM master structure
`vc_id`	Virtual circuit ID
`op`	Indicates start or stop operation

Output Parameters

None

Return Values

NSM_ERR_VC_ID_NOT_FOUND when the virtual circuit is not found

NSM_MPLS_BFD_VCCV_ERR_SESS_EXISTS when the BFD VCCV session already exists

NSM_MPLS_BFD_VCCV_ERR_NOT_CONFIGURED when the BFD VCCV session configuration fails

NSM_MPLS_BFD_VCCV_ERR_SESS_NOT_EXISTS when the BFD BCCV session does not exists

NSM_FAILURE when a generic error occurs

NSM_SUCCESS when the function succeeds

CHAPTER 6 Network Services Module

This chapter describes the interaction between the Network Service Module (NSM) Static Route Module and the BFD module. Included in this chapter is an overview covering the interaction between the NSM and BFD modules, information about the interfaces between these two modules along with the messages exchanged between them, and a description of the static route command functions that support BFD.

Overview

BFD for NSM Static Route module is a core module that handles static route functionality based on BFD session updates. NSM searches all Routing Information Base (RIB) entries upon receiving a BFD session event, which uses the specified nexthop address and outgoing interface from its routing table entries. The nexthop could match any one of the multiple nexthops stored in the RIB entry, which may be in either the active or inactive state. In each of the previously-matched nexthop entries that correspond to a static route (RIB type IPI_ROUTE_STATIC), a flag is set to indicate BFD reachability (NEXTHOP_FLAG_BFD_INACTIVE), based on the received session event.

When a “session down” event is received, the NEXTHOP_FLAG_BFD_INACTIVE flag is set. When a “session up” event is received, the NEXTHOP_FLAG_BFD_INACTIVE flag is cleared. The RIB entry is further processed to determine a path with an alternate nexthop or to modify the validity of the static route. If an alternate nexthop entry is available, the RIB entry is modified with the new nexthop. This may also result in static route being recursively resolved through a different nexthop and interface. If there are no valid nexthops available, the RIB entry is marked inactive and deleted from Forwarding Information Base (FIB). The static route status is updated to the client protocols into which it has been re-distributed.

The nexthop validity check in NSM verifies the NEXTHOP_FLAG_BFD_INACTIVE flag along with other conditions to determine the nexthop status. If NEXTHOP_FLAG_BFD_INACTIVE is set, the nexthop is considered as invalid and the next best nexthop is chosen for the given static route RIB entry. In case of multihop static routes, the recursively-resolved directly-connected gateway address and outgoing interface are already stored in the nexthop structure (rtype, rifindex and rgate). They are passed as session parameters towards BFD. Thus, during the RIB entry lookup, it is also necessary to match the BFD session details with the recursive nexthop data. BFD for NSM manages the following tasks:

- Initiating session towards BFD in OAMD for static route nexthops
- Handling the static route function calls based on the callback trigger received from BFD

NSM triggers the creation of a BFD session for the interface and nexthop on which a static route is configured. BFD creates a BFD session on an interface for the given nexthop and aids in detecting a data plane nexthop failure. Only one BFD session is created for all static routes with the same nexthop. BFD session modifications are conveyed from the BFD server back to the NSM client in the form of events. NSM processes these events to decide and mark the corresponding static routes’ validity.

Interfaces and Messages

This section describes the interfaces between BFD and NSM and provides a list of the messages that are exchanged between these modules.

Interfaces

NSM interacts with the base module using a BFD client to program BFD for related functionality. The BFD client interacts with external modules including the NSM Management and NSM Static interfaces.

BFD NSM Management Interface

The BFD NSM Interface is the interface to the NSM module for forwarding plane liveliness and nexthop data plane failure detection of static routes. Typical operations supported are to create, delete or update sessions. It is also required to specify any BFD Graceful restart parameters for a session.

BFD NSM Static Interface

The BFD NSM Static Interface provides an interface to the NSM module for detecting the static route nexthop data-plane failure. The typical operations supported are to create, delete or update sessions.

Messages

The following messages are exchanged between the BFD client and the BFD NSM Management Interface:

- BFD_SESSION_ADD
- BFD_SESSION DEL
- BFD_SESSION_REM
- BFD_SESSION_DISABLE
- BFD_SESSION_NO_DISABLE

The following messages are exchanged between the BFD client and the BFD NSM Static Interface:

- NSM_BFD_STATIC_SESSION_INVALID_OP
- NSM_BFD_STATIC_SESSION_ADD
- NSM_BFD_STATIC_SESSION_DEL
- NSM_BFD_STATIC_SESSION_REM
- NSM_BFD_STATIC_SESSION_ADMIN_DEL

Command API

The NSM module uses the following static route functions for BFD. The functions defined in this section are called by the commands described in the *Bidirectional Forwarding Detection Command Reference*.

Function	Application
nsm_ipv4_if_bfd_static_set	Configures BFD static route support on an interface for address family IPv4 for a given VR
nsm_ipv4_if_bfd_static_unset	Removes BFD static route support on an interface for address family IPv4 for a given VR

Function	Application
nsm_ipv4_if_bfd_static_set_all	Configures BFD static route support on all interfaces for address family IPv4 for a given VR
nsm_ipv4_if_bfd_static_unset_all	Removes BFD static route support on all interfaces for address family IPv4 for a given VR
nsm_ipv6_if_bfd_static_set	Configures BFD static route support on an interface for address family IPv6 for a given VR
nsm_ipv6_if_bfd_static_unset	Removes BFD static route support on an interface for address family IPv6 for a given VR
nsm_ipv6_if_bfd_static_set_all	Configures BFD static route support on all interfaces for address family IPv6 for a given VR
nsm_ipv6_if_bfd_static_unset_all	Removes BFD static route support on all interfaces for address family IPv6 for a given VR

nsm_ipv4_if_bfd_static_set

This function configures BFD static route support on an interface for address family IPv4 for a given VR.

Syntax

```
int
nsm_ipv4_if_bfd_static_set (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. The default value is 0. For a non-VR implementation, pass 0 for this parameter.
`ifname`	Name of the interface.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_IF_NOT_EXIST when the given interface does not exist or there is no NSM interface associated with the given interface

NSM_API_SET_ERR_STATIC_BFD_SET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv4_if_bfd_static_unset

This function removes BFD static route support on an interface for address family IPv4 for a given VR.

Syntax

```
int
nsm_ipv4_if_bfd_static_unset (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. The default value is 0. For a non-VR implementation, pass 0 for this parameter.
`ifname`	Name of the interface.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_IF_NOT_EXIST when the interface given does not exist, or there is no NSM interface associated with the given interface

NSM_API_SET_ERR_STATIC_BFD_UNSET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv4_if_bfd_static_set_all

This function configures BFD static route support on all interfaces for address family IPv4 for a given VR.

Syntax

```
int
nsm_ipv4_if_bfd_static_set_all (u_int32_t vr_id)
```

Input Parameters

`vr_id`	Virtual router ID. The default value is 0. For a non-VR implementation, pass 0 for this parameter.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_IF_NOT_EXIST when the interface given does not exist, or there is no NSM interface associated with the given interface

NSM_API_SET_ERR_STATIC_BFD_SET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv4_if_bfd_static_unset_all

This function removes BFD static route support on all interfaces for address family IPv4 for a given VR.

Syntax

```
int
nsm_ipv4_if_bfd_static_unset_all (u_int32_t vr_id)
```

Input Parameters

`vr_id`	Virtual router ID. The default value is 0. For a non-VR implementation, pass 0 for this parameter.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_STATIC_BFD_SET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv6_if_bfd_static_set

This function configures BFD static route support on an interface for address family IPv6 for a given VR.

Syntax

```
int
nsm_ipv6_if_bfd_static_set (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Name of the interface.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_IF_NOT_EXIST when the interface given does not exist, or there is no NSM interface associated with the given interface

NSM_API_SET_ERR_STATIC_BFD_SET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv6_if_bfd_static_unset

This function removes BFD static route support on an interface for address family IPv6 for a given VR.

Syntax

```
int
nsm_ipv6_if_bfd_static_unset (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. The default value is 0. For a non-VR implementation, pass 0 for this parameter.
`ifname`	Name of the interface.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_IF_NOT_EXIST when the interface given does not exist, or there is no NSM interface associated with the given interface

NSM_API_SET_ERR_STATIC_BFD_UNSET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv6_if_bfd_static_set_all

This function configures BFD static route support on all interfaces for address family IPv6 for a given VR.

Syntax

```
int
nsm_ipv6_if_bfd_static_set_all (u_int32_t vr_id)
```

Input Parameters

`vr_id`	Virtual router ID. The default value is 0. For a non-VR implementation, pass 0 for this parameter.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_STATIC_BFD_SET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

nsm_ipv6_if_bfd_static_unset_all

This function removes BFD static route support on all interfaces for address family IPv6 for a given VR.

Syntax

```
int
nsm_ipv6_if_bfd_static_unset_all (u_int32_t vr_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.

Output Parameters

None

Return Values

NSM_API_SET_ERROR when a generic error occurs

NSM_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

NSM_API_SET_ERR_STATIC_BFD_UNSET when the requested setting is already set

NSM_API_SET_ERR_VRF_NOT_EXIST when the virtual routing/forwarding instances does not exist

NSM_API_SET_SUCCESS when the function succeeds

CHAPTER 7 Border Gateway Protocol

This chapter describes the interaction between Border Gateway Protocol (BGP) and BFD. Included in this chapter is an overview covering the interaction between BGP and BFD, information about the interfaces between these two protocols along with the messages exchanged between them, and a description of the BGP command functions that support BFD.

Overview

The BFD module supports the BGP protocol module that is part of the ZebOS-XP protocol suite.

The BFD BGP module is the plug-in module for the BGP protocol to support BFD capabilities for both IPv4 and IPv6. It manages the following tasks in addition to initialization and termination of the BFD client module:

- BFD session handling for the BGP peer-reachability detection
- Callback registration for the BFD session-event handling

The BFD BGP module employs BFD as the peer-reachability detection protocol for both iBGP (Internal BGP) and eBGP (External BGP) peers. It adds a BFD session when a BGP peer reaches the ESTABLISHED state and deletes it if the peer moves down from the ESTABLISHED state.

A BFD client session key is generated using a combination of the outgoing interface index as the BFD session interface index, the source IP address of the BGP TCP session as the BFD source address, and the destination IP address of the BGP TCP session as the BFD destination address.

BGP BFD registers callback functions for BGP to respond to BFD session-down events and registers a BGP peer keep-alive expiration event via a session-down callback. It also registers callbacks for events when BFD is enabled or disabled in the context of a specific user VR.

An ADMIN_DOWN flag for the BFD session-down request is set if the session-down request is triggered by an operator command.

Interfaces and Messages

This section describes the interfaces between BFD and BGP and provides a list of the messages that are exchanged between these two protocols.

Interfaces

NSM interacts with the base module using a BFD client to program BFD for related functionality. The BFD client interacts with external modules including the BFD BGP management interface.

BFD BGP Management Interface

The BFD BGP management interface provides the interface to the NSM module for forwarding plane liveliness detection of static routes. The typical operations supported are to create, update or delete sessions. It is also required to specify any BFD graceful restart parameters for a session.

Messages

The following messages are exchanged between the BFD client and the BGP Management Interface:

- BFD_MSG_SESSION_ADD
- BFD_MSG_SESSION_DELETE
- BFD_MSG_SESSION_UP
- BFD_MSG_SESSION_DOWN
- BFD_MSG_SESSION_ERROR
- BFD_MSG_SESSION_ATTR_UPDATE

Command API

The BGP protocol module uses the following functions for BFD:

Function	Application
bgp_peer_bfd_set	Sets the BFD fall-over check for a specified peer
bgp_peer_bfd_unset	Unsets the BFD fall-over check for a specified peer

bgp_peer_bfd_set

This function sets the BFD fall-over check for a specified peer.

Syntax

```
int
bgp_peer_bfd_set (u_int32_t vr_id, vrf_id_t vrf_id, char *peer_str, bool_t mh)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`vrf_id`	Virtual routing forwarder ID.
`peer_str`	Character string that identifies the peer.
`mh`	Indicates whether the peer is a multi-hop (PAL_TRUE) or single-hop (PAL_FALSE) peer.

Output Parameters

None

Return Values

BGP_API_SET_ERR_INVALID_BGP when BGP instance is invalid

BGP_API_SET_ERR_PEER_MALFORMED_ADDRESS when the peer address is malformed

BGP_API_SET_ERR_PEER_SELF_ADDRESS when the peer address is the same as the interface address

BGP_API_SET_ERR_PEER_UNINITIALIZED when the peer has not been initialized

BGP_API_SET_ERR_PEER_DUPLICATE when the peer address is a duplicate

BGP_API_SET_SUCCESS when the function succeeds

bgp_peer_bfd_unset

This function unsets the BFD fall-over check for a specified peer.

Syntax

```
int
bgp_peer_bfd_unset (u_int32_t vr_id, vrf_id_t vrf_id, char *peer_str)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`vrf_id`	Virtual routing forwarder ID.
`peer_str`	Character string that identifies the peer.

Output Parameters

None

Return Values

BGP_API_SET_ERR_INVALID_BGP when BGP instance is invalid

BGP_API_SET_ERR_PEER_MALFORMED_ADDRESS when the peer address is malformed

BGP_API_SET_ERR_PEER_SELF_ADDRESS when the peer address is the same as the interface address

BGP_API_SET_ERR_PEER_UNINITIALIZED when the peer has not been initialized

BGP_API_SET_ERR_PEER_DUPLICATE when the peer address is a duplicate

BGP_API_SET_SUCCESS when the function succeeds

CHAPTER 8 Intermediate System To Intermediate System

This chapter describes the interaction between the Intermediate System To Intermediate System (IS-IS) and BFD. Included in this chapter is an overview covering the interaction between IS-IS and BFD, information about the interfaces between these two protocols along with the messages exchanged between them, and a description of the IS-IS command functions that support BFD.

Overview

The BFD module supports the IS-IS protocol module that is part of the ZebOS-XP protocol suite. The BFD IS-IS is the plug-in module for the IS-IS protocol to support BFD capabilities for both IPv4 and IPv6. It manages the following tasks in addition to initialization and termination of the BFD client module:

- BFD session handling for IS-IS neighbor-reachability detection
- Callback registration for BFD session-event handling

The BFD IS-IS module employs BFD as the neighbor-reachability detection protocol for both shared links and point-to-point links. It adds a BFD session when an IS-IS neighbor is in the Full state, except on a shared link. A session is added only when either the neighbor or itself is Designated Intermediate System (DIS) on a shared link to prevent an unnecessary full mesh of the BFD session on shared media. It deletes the BFD session when the neighbor goes below the Full state.

A BFD client session key is generated using a combination of the IS-IS outgoing-interface index as the BFD session-interface index, and the IS-IS outgoing-interface primary-IP address. They are advertised by the first IP interface address TLV in the hello packet as the BFD source address and the IS-IS neighbor primary IP address, which are advertised by the first IP interface address TLV in the neighbor's hello packet as the BFD destination address.

BFD IS-IS registers callback functions for IS-IS to respond to BFD session-down events and registers an IS-IS neighbor hold-timer-expired event via a session-down callback. An ADMIN_DOWN flag for a BFD session-down request is set if the session-down request is triggered by an operator command.

Interfaces and Messages

This section describes the interfaces between BFD and IS-IS and provides a list of the messages that are exchanged between these two protocols.

Interfaces

NSM interacts with the base module using a BFD client to program BFD for related functionality. The BFD client interacts with external modules including the BFD IS-IS management interface.

BFD IS-IS Management Interface

The BFD IS-IS management interface provides the interface to the NSM module for the forwarding plane liveliness detection of static routes. The typical operations supported are to create, update or delete sessions. It is also required to specify any BFD graceful restart parameters for a session.

Messages

The following messages are exchanged between the BFD client and the BFD IS-IS Management Interface:

- BFD_MSG_SESSION_ADD
- BFD_MSG_SESSION_DELETE
- BFD_MSG_SESSION_UP
- BFD_MSG_SESSION_DOWN
- BFD_MSG_SESSION_ERROR
- BFD_MSG_SESSION_ATTR_UPDATE

Command API

The IS-IS protocol module uses the following functions for BFD:

Function	Application
isis_if_bfd_set	Sets the BFD fall-over check for neighbors on specified interface
isis_if_bfd_unset	Sets the BFD fall-over check for neighbors on specified interface
isis_if_bfd_disable_set	Disables the BFD fall-over check for neighbors on specified interface
isis_if_bfd_disable_unset	Unsets the disable flag of BFD fall-over check for neighbors on specified interface
isis_bfd_all_interfaces_set	Sets the BFD fall-over check for all the interfaces under a specified process
isis_bfd_all_interfaces_unset	Unsets the BFD fall-over check for all the interfaces under a specified process

isis_if_bfd_set

This function sets the BFD fall-over check for neighbors on specified interface. It enables BFD on an interface.

Syntax

```
int
isis_if_bfd_set (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

ISIS_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

ISIS_API_SET_SUCCESS when the function succeeds

isis_if_bfd_unset

This function unsets the BFD fall-over check for neighbors on a specified interface. It disables BFD on an interface.

Syntax

```
int
isis_if_bfd_unset (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifnam`	Interface name.

Output Parameters

None

Return Values

SIS_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

ISIS_API_SET_SUCCESS when the function succeeds

isis_if_bfd_disable_set

This function disables the BFD fall-over check for neighbors on specified interface. It sets BFD to disable on an interface.

Syntax

```
int
isis_if_bfd_disable_set (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

ISIS_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

ISIS_API_SET_SUCCESS when the function succeeds

isis_if_bfd_disable_unset

This function unsets the disable flag of BFD fall-over check for neighbors on a specified interface. It unsets the BFD disable parameter on an interface.

Syntax

```
int
isis_if_bfd_disable_unset (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

SIS_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

ISIS_API_SET_SUCCESS when the function succeeds

isis_bfd_all_interfaces_set

This function sets the BFD fall-over check for all the interfaces under a specified process.

Syntax

```
int
isis_bfd_all_interfaces_set (u_int32_t vr_id, char *tag)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`tag`	String that identifies a particular IS-IS instance.

Output Parameters

None

Return Values

ISIS_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

ISIS_API_SET_ERR_INSTANCE_NOT_EXIST when the instance does not exist

ISIS_API_SET_SUCCESS when the function succeeds

isis_bfd_all_interfaces_unset

This function unsets the BFD fall-over check for all the interfaces under a specified process.

Syntax

```
int
isis_bfd_all_interfaces_unset (u_int32_t vr_id, char *tag)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`tag`	String that identifies a particular IS-IS instance.

Output Parameters

None

Return Values

ISIS_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

ISIS_API_SET_ERR_INSTANCE_NOT_EXIST when the instance does not exist

ISIS_API_SET_SUCCESS when the function succeeds

CHAPTER 9 Open Shortest Path First

This chapter describes the interaction between Open Shortest Path First (OSPF) and BFD. Included in this chapter is an overview covering the interaction between the OSPF and BFD modules, information about the interfaces between these two protocols along with the messages exchanged between them, and a description of the OSPFv2 command functions that support BFD.

Overview

The BFD module supports the OSPFv2 protocol modules that is part of the ZebOS-XP protocol suite.

BFD OSPFv2 Module

The BFD OSPFv2 module is the plug-in module for the OSPFv2 to support BFD capability. It manages the following tasks in addition to initialization and termination of the BFD client module:

- BFD session handling for OSPF neighbor reachability detection
- Callback registration for BFD session-event handling

The OSPFv2 BFD module employs BFD as the neighbor-reachability detection protocol for both standard shared links and OSPF virtual links. It adds a BFD session when an OSPF neighbor goes beyond the OSPF two-way state, thereby avoiding an unnecessary full mesh of the BFD session on shared links. The BFD session is deleted when the neighbor is below or equal to the two-way state.

A BFD client session key is generated with a combination of the OSPF outgoing-interface index as the BFD session-interface index, the OSPF outgoing interface IP address as the BFD source address, and the OSPF neighbor IP address as the BFD destination address. The multi-hop flag is also set when the neighbor is on an OSPF virtual link.

BFD OSPF registers callback functions for OSPF to respond to BFD session down events, and registers an OSPF kill-neighbor event via a session-down callback. It also registers callbacks for events when BFD is enabled or disabled in the context of a specific user VR. An ADMIN_DOWN flag for a BFD session-down request is set if the session-down request is triggered by an operator command.

Interfaces and Messages

This subsection describes the interfaces between BFD and OSPF and provides a list of the messages that are exchanged between these two protocols.

Interfaces

NSM interacts with the base module using a BFD client to program BFD for related functionality. The BFD client interacts with external modules including the BFD OSPFv2 management interface.

BFD OSPFv2 Management Interface

The BFD OSPFv2 management interface is the interface to the NSM module for forwarding plane liveliness detection of static routes. Typical operations supported are to create, update or delete sessions. It is also required to specify any BFD graceful restart parameters for a session.

Messages

The following messages are exchanged between the BFD client and the BFD OSPF management interfaces:

- BFD_MSG_SESSION_ADD
- BFD_MSG_SESSION_DELETE
- BFD_MSG_SESSION_UP
- BFD_MSG_SESSION_DOWN
- BFD_MSG_SESSION_ERROR
- BFD_MSG_SESSION_ATTR_UPDATE

OSPFv2 BFD Command API

The OSPFv2 protocol module uses the following functions for BFD:

Function	Application
ospf_if_bfd_set	Sets the BFD fall-over check for neighbors on a specified interface
ospf_if_bfd_unset	Unsets the BFD fall-over check for neighbors on a specified interface
ospf_if_bfd_disable_set	Disables the BFD fall-over check for neighbors on a specified interface
ospf_if_bfd_disable_unset	Unsets the disable flag of BFD fall-over check for neighbors on a specified interface
ospf_bfd_all_interfaces_set	Sets the BFD fall-over check for all the neighbors under a specified process
ospf_bfd_all_interfaces_unset	Unsets the BFD fall-over check for all the neighbors under a specified process
ospf_vlink_bfd_set	Sets the BFD fall-over check for a specified virtual-link neighbor
ospf_vlink_bfd_unset	Unsets the BFD fall-over check for a specified virtual-link neighbor

ospf_if_bfd_set

This function sets the BFD fall-over check for neighbors on a specified interface. It enables BFD on an interface.

Syntax

```
int
ospf_if_bfd_set (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_SUCCESS when the function succeeds

ospf_if_bfd_unset

This function unsets the BFD fall-over check for neighbors on a specified interface. It disables BFD on an interface.

Syntax

```
int
ospf_if_bfd_unset (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_SUCCESS when the function succeeds

ospf_if_bfd_disable_set

This function disables the BFD fall-over check for neighbors on a specified interface. It sets BFD to disable on an interface. This function takes precedence over a global BFD configuration.

Syntax

```
int
ospf_if_bfd_disable_set (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_SUCCESS when the function succeeds

ospf_if_bfd_disable_unset

This function unsets the disable flag of BFD fall-over check for neighbors on a specified interface. It unsets the BFD disable parameter on an interface. This function takes precedence over a global BFD configuration.

Syntax

```
int
ospf_if_bfd_disable_unset (u_int32_t vr_id, char *ifname)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`ifname`	Interface name.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_SUCCESS when the function succeeds

ospf_bfd_all_interfaces_set

This function sets the BFD fall-over check for all the interfaces under a specified process.

Syntax

```
int
ospf_bfd_all_interfaces_set (u_int32_t vr_id, int proc_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for affected interfaces.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_ERR_PROCESS_ID_INVALID when the process ID is not valid

OSPF_API_SET_ERR_PROCESS_NOT_EXIST when there is no active OSPF process with this ID

OSPF_API_SET_SUCCESS when the function succeeds

ospf_bfd_all_interfaces_unset

This function unsets the BFD fall-over check for all the interfaces under a specified process.

Syntax

```
int
ospf_bfd_all_interfaces_unset (u_int32_t vr_id, int proc_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for affected interfaces.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_ERR_PROCESS_ID_INVALID when the process ID is not valid

OSPF_API_SET_ERR_PROCESS_NOT_EXIST when there is no active OSPF process with this ID

OSPF_API_SET_SUCCESS when the function succeeds

ospf_vlink_bfd_set

This function sets the BFD fall-over check for a specified virtual-link neighbor. It enables BFD for a virtual-link neighbor.

Syntax

```
int
ospf_vlink_bfd_set (u_int32_t vr_id, int proc_id,
                    struct pal_in4_addr area_id,
                    struct pal_in4_addr peer_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for affected interfaces.
`area_id`	Area ID in which the virtual-link neighbor exists in the form of an IPv4 address in dotted-decimal notation.
`peer_id`	ID of the virtual-link neighbor in the form of an IPv4 address in dotted decimal notation.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_ERR_PROCESS_ID_INVALID when the process ID is not valid

OSPF_API_SET_ERR_PROCESS_NOT_EXIST when there is no active OSPF process with this ID

OSPF_API_SET_ERR_VLINK_NOT_EXIST when a virtual link to the virtual-link neighbor does not exist

OSPF_API_SET_SUCCESS when the function succeeds

ospf_vlink_bfd_unset

This function unsets the BFD fall-over check for a specified virtual-link neighbor. It disables BFD for a virtual-link neighbor.

Syntax

```
int
ospf_vlink_bfd_unset (u_int32_t vr_id, int proc_id,
                      struct pal_in4_addr area_id,
                      struct pal_in4_addr peer_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for affected interfaces.
`area_id`	Area ID in which the virtual-link neighbor exists in the form of an IPv4 address in dotted-decimal notation.
`peer_id`	ID of the virtual-link neighbor in the form of an IPv4 address in dotted decimal notation.

Output Parameters

None

Return Values

OSPF_API_SET_ERR_VR_NOT_EXIST when there is no active VR instance

OSPF_API_SET_ERR_PROCESS_ID_INVALID when the process ID is not valid

OSPF_API_SET_ERR_PROCESS_NOT_EXIST when there is no active OSPF process with this ID

OSPF_API_SET_ERR_VLINK_NOT_EXIST when a virtual link to the virtual-link neighbor does not exist

OSPF_API_SET_SUCCESS when the function succeeds

CHAPTER 10 Routing Information Protocol

This chapter describes the interaction between the Routing Information Protocol (RIP) and BFD. Included in this chapter are the following:

- Overview covering the interaction between the RIP and BFD modules
- Information about the interfaces between these two protocols
- Messages exchanged between these two protocols
- Description of the RIP command functions that support BFD

Overview

The BFD module supports the RIP protocol module that is part of the ZebOS-XP protocol suite by monitoring RIP adjacency with its neighbors using BFD. This feature is used to contribute to finding alternate paths when a neighbor is down.

RIP uses the timeout of prefixes for a particular neighbor to identify whether a neighbor is inactive. By default, the timeout is 180 seconds. If the next-hop router is inactive, the RIP router continues to broadcasts prefixes for up to 180 seconds, which can lead to data loss. To avoid this type of data loss, BFD is used to detect the neighbor path failure within a sub second.

The BFD protocol module runs as a separate process (as part of `oamd`) in the ZebOS-XP architecture model. It runs in conjunction with the routing protocols. BFD and RIP interact using the client server mechanism with BFD being the server and RIP and the other protocol modules being the BFD clients.

The other routing protocol, such as OSPF or BGP, maintain a neighbor's adjacency state information. RIP maintains a list of configured neighbors for all neighbors for which BFD is enabled. RIP uses the update packets it receives from its neighbors to update this list. The neighbor addresses in the list are considered for establishing the BFD sessions.

RIP registers with BFD, which triggers the creation of a BFD session for a neighbor. BFD then creates a session for the given neighbor. If the neighbor goes down or becomes inactive, RIP receives a down event through a BFD call-back handler. For the down event, RIP executes the RIP timer handler for all routes learned from the given neighbor. The RIP timeout handler function executes after the expiration of the RIP timer and marks the routes of inactive neighbors as invalid.

If an established BFD session for a neighbor fails, and a route is received from that neighbor with a next hop address that is not the address of neighbor, the route lingers until it times out (after 180 seconds). To avoid this lingering route problem, all of the routes from the neighbor corresponding to the failed BFD session are timed out.

When RIP initiates a BFD session with the following session parameters:

- Outgoing interface index
- IPv4 address of the outgoing interface
- Neighbor address

These correspond to the following:

- BFD session interface index
- Session outgoing interface
- Session neighbor

Interfaces and Messages

This subsection describes the interfaces between BFD and RIP and provides a list of the messages that are exchanged between these two protocols.

Interfaces

RIP is the client to the BFD. When communication is required between these two modules, RIP directly interacts with BFD using the BFD client.

BFD RIP Management Interface

The BFD RIP management interface provides forwarding plane liveliness detection of RIP neighbors. Typically, this module creates, deletes or updates sessions. It is also required to specify any BFD Graceful restart parameters for a session.

Messages

The following messages are exchanged between the BFD client and the BFD RIP Management Interface:

- BFD_MSG_SESSION_ADD
- BFD_MSG_SESSION_DELETE
- BFD_MSG_SESSION_UP
- BFD_MSG_SESSION_DOWN
- BFD_MSG_SESSION_ERROR

RIP and BFD Command API

The RIP protocol module uses the following functions for BFD:

Function	Application
rip_bfd_all_interfaces_set	Sets the BFD fall-over check for all the interfaces under a specified process
rip_bfd_all_interfaces_unset	Unsets the BFD fall-over check for all the interfaces under a specified process
rip_bfd_neighbor_set	Sets the BFD fall-over check for a specific neighbor under a specified process
rip_bfd_neighbor_unset	Unsets the BFD fall-over check for a specific neighbor under a specified process
rip_bfd_debug_set	Enables debugging for RIP BFD
rip_bfd_debug_unset	Disables debugging for RIP BFD

rip_bfd_all_interfaces_set

This function sets the BFD fall-over check for all the interfaces under a specified process.

This function is called by the `bfd all-interfaces` command.

Syntax

```
int
rip_bfd_all_interfaces_set (u_int32_t vr_id, int instance);
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`instance`	The instance identifier.

Output Parameters

None

Return Values

RIP_API_SET_ERR_VR_NOT_EXIST when the virtual router does not exist

RIP_API_SET_ERR_PROCESS_NOT_EXIST when there is no active RIP process with this ID

RIP_API_SET_ERR_BFD_CONF_SET when RIP BFD is already enabled

RIP_API_SET_SUCCESS when the function succeeds

rip_bfd_all_interfaces_unset

This function unsets the BFD fall-over check for all the interfaces under a specified process.

This function is called by the `no bfd all-interfaces` command.

Syntax

```
int
rip_bfd_all_interfaces_unset (u_int32_t vr_id, int instance);
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`instance`	The instance identifier.

Output Parameters

None

Return Values

RIP_API_SET_ERR_VR_NOT_EXIST when the virtual router does not exist

RIP_API_SET_ERR_PROCESS_NOT_EXIST when there is no active RIP process with this ID

RIP_API_SET_ERR_BFD_CONF_UNSET when RIP BFD is already disabled

RIP_API_SET_SUCCESS when the function succeeds

rip_bfd_neighbor_set

This function sets the BFD fall-over check for a specific neighbor under a specified process.

This function is called by the `neighbor A.B.C.D fall-over bfd` command.

Syntax

```
int
rip_bfd_neighbor_set (struct pal_in4_addr *temp_nbr, u_int32_t vr_id, int instance);
```

Input Parameters

`temp_nbr`	Neighbor address in an IPv4 address format.
`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`instance`	The instance identifier.

Output Parameters

None

Return Values

RIP_API_SET_ERR_VR_NOT_EXIST when the virtual router does not exist

RIP_API_SET_ERR_PROCESS_NOT_EXIST when there is no active RIP process with this ID

RIP_API_SET_ERR_BFD_NEIGHBOR_INVALID when the BFD neighbor is invalid

RIP_API_SET_ERR_BFD_CONF_SET when RIP BFD is already enabled

RIP_API_SET_SUCCESS when the function succeeds

rip_bfd_neighbor_unset

This function unsets the BFD fall-over check for a specific neighbor under a specified process.

This function is called by the `no neighbor A.B.C.D fall-over bfd` command.

Syntax

```
int
rip_bfd_neighbor_unset (struct pal_in4_addr *nbr, u_int32_t vr_id, int instance);
```

Input Parameters

`nbr`	Neighbor address in an IPv4 address format.
`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation
`instance`	The instance identifier.

Output Parameters

None

Return Values

RIP_API_SET_ERR_VR_NOT_EXIST when the virtual router does not exist

RIP_API_SET_ERR_PROCESS_NOT_EXIST when there is no active RIP process with this ID

RIP_API_SET_ERR_BFD_NEIGHBOR_INVALID when the BFD neighbor is invalid

RIP_API_SET_SUCCESS when the function succeeds

RIP_API_SET_ERR_BFD_CONF_UNSET when RIP BFD is already disabled

rip_bfd_debug_set

This function enables debugging for RIP BFD.

This function is called by the `debug rip bfd` command.

Syntax

```
void
rip_bfd_debug_set (struct rip_master *rm);
```

Input Parameters

`rm`	Pointer to the RIP master

Output Parameters

None

Return Values

LIB_API_ERROR when the debugging fails

LIB_API_SUCCESS when the debugging succeeds

rip_bfd_debug_unset

This function disables debugging for RIP BFD.

This function is called by the `no debug rip bfd` command.

Syntax

```
void
rip_bfd_debug_unset (struct rip_master *rm);
```

Input Parameters

`rm`	Pointer to the RIP master

Output Parameters

None

Return Values

LIB_API_ERROR when the debugging fails

LIB_API_SUCCESS when the debugging succeeds

CHAPTER 11 BFD Authentication

This chapter describes the authentication support for the BFD module. Authentication provides a security mechanism for the BFD module. This helps mitigate threats on a BFD session from hackers and attackers.

Overview

There always is the possibility of an attacker or hacker of gaining control of a link between systems whenever a BFD session runs between two peers. When in control, a hacker can easily drop BFD packets and forward everything else, which will cause the link to be falsely declared down. In addition, a hacker can also forward only BFD packets and not anything else, which causes the link to be falsely declared up. Authentication helps prevent these attacks.

Authentication Types

The following authentication types provides a level of security varies based on the desired type chosen. That is, Simple password is the weakest type and meticulous SHA1 is the strongest.

- Simple password
- Keyed/Meticulous MD5
- Keyed/Meticulous SHA1

Software Design

BFD commands enable authentication on both the interface and configure level for multihop IPv4 and IPv6 sessions. Refer to the *Bidirectional Forwarding Detection Command Reference* for more information about these commands.

Authentication commands are stored in a config file by using a “write” command. Whenever authentication is enabled on a session, it starts sending the authentication header in BFD packet. Whenever a BFD packet is received with an authentication header, it is processed and authenticated.

The key-chain module is used for multi-key BFD support. When multiple keys are configured by using the key-chain command, the selection of a key ID among multiple keys is based on the time in which the authentication key is active.

The encoding and decoding of the authentication section in the BFD control packets is based on the configured authentication type. In addition, updating and validating the sequence number on an auth-section for BFD packets is transmitted and received on types MD5 and SHA1, respectively.

Command APIs

The functions defined in this chapter are called by the BFD authentication commands.

Function	Application
bfd_construct_auth	Encodes an authentication section for a BFD packet.
bfd_proto_auth_set	Sets the authentication info on interface having sessions
bfd_proto_auth_unset	Unsets the authentication info on interface having sessions
bfd_proto_multihop_auth_set	Sets the authentication information for a multihop session.
bfd_proto_multihop_auth_unset	Unsets the authentication information from a multihop session.
bfd_construct_mpls_packet	Constructs the BFD packet for both an MPLS- and VCCV-related BFD sessions.
bfd_construct_packet	Creates a BFD packet.
bfd_construct_packet6	Creates a BFD packet.
bfd_avl_if_config_write	Saves the running configured parameters at the interface level.
bfd_mh_config_write_for_if	Saves the running multihop authentication configuration at the configure level.
bfd_auth_process_validate	Decodes and validates an authentication section on a BFD packet.

bfd_construct_auth

This function encodes an authentication section for a BFD packet.

Syntax

```
int
bfd_construct_auth (struct bfd_session_fwd *sess, u_char *bfd_pkt_buf)
```

Input Parameters

`*sess`	BFD session forward entry.
`*bfd_pkt_buf`	BFD packet buffer.

Output Parameters

None

Return Values

BFD_FALSE on failure

BFD_SUCCESS on success

BFD_AUTH_NOT_ENABLED

BFD_INVALID_AUTH_TYPE

BFD_INVALID_KEY

BFD_MD5_AUTH_SEC_LEN

BFD_SHA1_AUTH_SEC_LEN

bfd_proto_auth_set

This function sets the authentication information on an interface having sessions.

Syntax

```
int
bfd_proto_auth_set (u_int32_t vr_id, s_int32_t proc_id, char *ifname, \
                    char *auth_type_str,u_int32_t key_id,char *key_str,\
                    char *key_chain)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.
`*auth_type_str`	Authentication string type.
`key_id`	Key identifier.
`*key_str`	Key string.
`*key_chain`	Key chain.

Output Parameters

None

Return Values

BFD_FALSE on failure

BFD_SUCCESS on success

BFD_API_INVALID_AUTH_TYPE

BFD_API_INVALID_KEY_ID

BFD_API_SET_ERR_VR_NOT_EXIST

BFD_API_INSTANCE_NOT_FOUND

BFD_API_IF_NOT_FOUND

BFD_RET_TO_API_RET

bfd_proto_auth_unset

This function unsets the authentication information on an interface having sessions.

Syntax

```
s_int32_t
bfd_proto_auth_unset (u_int32_t vr_id, s_int32_t proc_id, uint32_t ifindex)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.

Output Parameters

None

Return Values

BFD_FALSE on failure

BFD_SUCCESS on success

BFD_API_INVALID_AUTH_TYPE

BFD_API_INVALID_KEY_ID

BFD_API_SET_ERR_VR_NOT_EXIST

BFD_API_INSTANCE_NOT_FOUND

BFD_API_IF_NOT_FOUND

BFD_RET_TO_API_RET

bfd_proto_multihop_auth_set

This function sets the authentication information for a multihop session.

Syntax

```
s_int32_t
bfd_proto_multihop_auth_set (u_int32_t vr_id, s_int32_t proc_id, \
                             u_int32_t ifindex, afi_t afi, void *addr,\
                             char *auth_type_str,u_int32_t key_id, \
                             char *key_str,char *key_chain)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.
`afi`	Address family identifier.
`*addr`	Address type.
`*auth_type_str`	Authentication string type.

`key_id`	Key identifier.
`*key_str`	Key string.
`*key_chain`	Key chain.

Output Parameters

None

Return Values

BFD_FALSE on failure

BFD_SUCCESS on success

BFD_API_INVALID_AUTH_TYPE

BFD_API_SET_ERR_VR_NOT_EXIST

BFD_API_INSTANCE_NOT_FOUND

BFD_API_IF_NOT_FOUND

BFD_API_MH_NOT_FOUND

BFD_RET_TO_API_RET

bfd_proto_multihop_auth_unset

This function unsets the authentication information from a multihop session.

Syntax

```
s_int32_t
bfd_proto_multihop_auth_unset (u_int32_t vr_id, s_int32_t proc_id, s_int32_t ifindex,
afi_t afi, void *addr)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.
`afi`	Address family identifier.
`*addr`	Address type.

Output Parameters

None

Return Values

BFD_FALSE on failure

BFD_SUCCESS on success

BFD_API_SET_ERR_VR_NOT_EXIST

BFD_API_INSTANCE_NOT_FOUND

BFD_API_IF_NOT_FOUND

BFD_API_MH_NOT_FOUND

BFD_RET_TO_API_RET

bfd_construct_mpls_packet

This function constructs the BFD packet for both an MPLS- and VCCV-related BFD sessions.

Syntax

```
void
bfd_construct_mpls_packet (struct bfd_session_fwd *sess)
```

Input Parameters

`*sess`	MPLS- or VCCV-related BFD session.

Output Parameters

None

Return Values

None

bfd_construct_packet

This function creates a BFD packet.

Syntax

```
void
bfd_construct_packet (struct bfd_session_fwd *sess)
```

Input Parameters

`*sess`	MPLS- or VCCV-related BFD session.

Output Parameters

None

Return Values

None

bfd_construct_packet6

This function creates a BFD packet.

Syntax

```
void
bfd_construct_packet6 (struct bfd_session_fwd *sess)
```

Input Parameters

`*sess`	MPLS- or VCCV-related BFD session.

Output Parameters

None

Return Values

None

bfd_avl_if_config_write

This function saves the running configured parameters at the interface level.

Syntax

```
s_int32_t
bfd_avl_if_config_write (void_t *avl_node_info,
                         void_t *arg1,
                         void_t *arg2)
```

Input Parameters

`*avl_node_info`	AVL node information.
`*arg1`	Argument 1
`*arg2`	Argument 2.

Output Parameters

None

Return Values

Return 0

bfd_mh_config_write_for_if

This function saves the running multihop authentication configuration at the configure level.

Syntax

```
s_int32_t
bfd_mh_config_write_for_if (struct cli *cli, struct bfd_interface *bif,
                            s_int32_t *write, char *str)
```

Input Parameters

`*cli`	CLI command
`*bif`	BFD interface
`*write`	Write information
`*str`	String information

Output Parameters

None

Return Values

Return 0

bfd_auth_process_validate

This function decodes and validates an authentication section on a BFD packet.

Syntax

```
bool_t
bfd_auth_process_validate (struct bfd_session_fwd *sess, u_char *bfd_pkt_buf)
```

Input Parameters

`*sess`	BFD authentication session.
`*bfd_pkt_buf`	BFD packet buffer.

Output Parameters

None

Return Values

BFD_FALSE validation failed

BFD_TRUE validation success

CHAPTER 12 Virtual Router

This chapter describes the interaction between the Virtual Router (VR) module and BFD. This chapter includes a description of BFD-related command functions.

Overview

The BFD Base Module interacts with the NSM to get interface and virtual router specific information. BFD functionality is available in the Privilege VR (PVR) context, which is the default VR context. It is also available in the non-PVR context, if the BFD is attached to the associated VR. BFD uses the VR-ID of a VR to determine which VR context to apply the BFD functionality.

Note: When handling client requests, the BFD determines the appropriate VR context to which to provide service using the VR-ID of the VR from which the request was sent.

BFD API for VR

The following BFD functions support virtual router API and are described in Chapter 5, *OAM for BFD and MPLS*:

Function	Application
nsm_mpls_bfd_api_fec_set	Sets the BFD fall-over check for the Forwarding Equivalency Class (FEC) of an label switched path (LSP) type (Label Discovery Protocol (LDP), Resource Reservation Protocol (RSVP) or Static)
nsm_mpls_bfd_api_fec_unset	Removes the BFD fall-over check for the FECs of an LSP-type
nsm_mpls_bfd_api_fec_disable_set	Disables the BFD fall-over check for the FEC or Trunk of an LSP- type
nsm_mpls_bfd_api_lsp_all_set	Sets the BFD fall-over check for all FECs of an LSP-type
nsm_mpls_bfd_api_lsp_all_unset	Removes the BFD fall-over check for all FECs of an LSP-type
nsm_mpls_bfd_api_vccv_trigger	Starts or stops the BFD VCCV session for a VC based on the specified operation

CHAPTER 13 BFD Command API

The BFD functions defined in this chapter are called by the BFD commands described in the *Bidirectional Forwarding Detection Command Reference*.

The following BFD command functions are described in this chapter:

Function	Application
bfd_add_user_session	Adds an IPv4 BFD user session
bfd_del_user_session	Deletes an IPv4 BFD user session
bfd_add_ipv6_user_session	Adds an IPv6 BFD user session
bfd_del_ipv6_user_session	Deletes an IPv6 BFD user session
bfd_echo_interval_set	Sets the BFD Echo mode transmission interval for all single-hop sessions on an interface
bfd_echo_interval_unset	Resets the BFD Echo mode transmission interval to its default value for all sessions on an interface
bfd_proto_interval_set	Sets minimum BFD transmission and reception intervals, and the value of the Hello multiplier for all sessions on an interface
bfd_proto_interval_unset	Resets BFD transmission and reception intervals and the Hello multiplier to their default values for all sessions on an interface
bfd_proto_multihop_interval_set	Sets multihop interval peer timer values, and the Hello multiplier for an IPv4 interface
bfd_proto_multihop_interval_unset	Resets multihop interval peer timers and the Hello multiplier for an IPv4 interface to their default values
bfd_protol_multihop_ipv6_interval_set	Sets multihop interval peer timer values and the Hello multiplier for an IPv6 interface
bfd_protol_multihop_ipv6_interval_unset	Resets multihop interval peer timers and the Hello multiplier for an IPv6 interface to their default values
bfd_echo_mode_set	Sets BFD to echo mode for all interfaces in a BFD process
bfd_echo_mode_unset	Removes BFD echo mode setting for all interfaces in a BFD process
bfd_proto_slow_timer_set	Sets a BFD slow timer for all interfaces in a BFD process
bfd_proto_slow_timer_unset	Removes a BFD slow timer for all interfaces in a BFD process
bfd_proto_slow_timer_set_interface	Sets the slow timer value for a particular interface
bfd_proto_slow_timer_unset_interface	Unsets the slow timer for a particular interface

bfd_add_user_session

This function adds an IPv4 BFD user session.

Syntax

```
s_int32_t
bfd_add_user_session (u_int32_t vr_id, s_int32_t proc_id,
                      struct pal_in4_addr *src_addr,
                      struct pal_in4_addr *dst_addr,
                      s_int32_t ifindex, uint32_t flags)
```

Input Parameters

`vr_id` Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.

`proc_id` Process ID for the BFD instance <0–65535>.

`src_addr` Source address for the session.

`dst_addr` Destination address for the session.

`ifindex` Interface index.

`flags` The flags define the properties of a session. They are:

`BFD_MSG_SESSION_FLAG_MH`
Multihop session.

`BFD_MSG_SESSION_FLAG_DC`
Session over Demand Circuit.

`BFD_MSG_SESSION_FLAG_PS`
Persistent Session.

`BFD_MSG_SESSION_FLAG_AD`
Session in Admin Down state.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_del_user_session

This function deletes an IPv4 BFD user session.

Syntax

```
s_int32_t
bfd_del_user_session (u_int32_t vr_id, s_int32_t proc_id,
                      struct pal_in4_addr *src_addr,
                      struct pal_in4_addr *dst_addr,
                      s_int32_t ifindex, u_int32_t flags)
```

Input Parameters

`vr_id` Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.

`proc_id` Process ID for the BFD instance <0–65535>.

`src_addr` Source address for the session.

`dst_addr` Destination address for the session.

`ifindex` Interface index.

`flags` The flags define the properties of a session. They are:

`BFD_MSG_SESSION_FLAG_MH`
Multihop session.

`BFD_MSG_SESSION_FLAG_DC`
Session over Demand Circuit.

`BFD_MSG_SESSION_FLAG_PS`
Persistent Session.

`BFD_MSG_SESSION_FLAG_AD`
Session in Admin Down state.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_add_ipv6_user_session

This function adds an IPv6 BFD user session.

Syntax

```
s_int32_t
bfd_add_ipv6_user_session (u_int32_t vr_id, s_int32_t proc_id,
                           struct pal_in6_addr *src_addr,
                           struct pal_in6_addr *dst_addr,
                           s_int32_t ifindex, u_int32_t flags)
```

Input Parameters

`vr_id` Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.

`proc_id` Process ID for the BFD instance <0–65535>.

`src_addr` Source address for the session.

`dst_addr` Destination address for the session.

`ifindex` Interface index.

`flags` The flags define the properties of a session. They are:

`BFD_MSG_SESSION_FLAG_MH`
Multihop session.

`BFD_MSG_SESSION_FLAG_DC`
Session over Demand Circuit.

`BFD_MSG_SESSION_FLAG_PS`
Persistent Session.

`BFD_MSG_SESSION_FLAG_AD`
Session in Admin Down state.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_INVALID_ADDR when the address is invalid

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_del_ipv6_user_session

This function deletes an IPv6 BFD user session.

Syntax

```
s_int32_t
bfd_del_ipv6_user_session (u_int32_t vr_id, s_int32_t proc_id,
                           struct pal_in6_addr *src_addr,
                           struct pal_in6_addr *dst_addr,
                           s_int32_t ifindex, u_int32_t flags)
```

Input Parameters

`vr_id` Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.

`proc_id` Process ID for the BFD instance <0–65535>.

`src_addr` Source address for the session.

`dst_addr` Destination address for the session.

`ifindex` Interface index.

`flags` The flags define the properties of a session. They are:

`BFD_MSG_SESSION_FLAG_MH`
Multihop session.

`BFD_MSG_SESSION_FLAG_DC`
Session over Demand Circuit.

`BFD_MSG_SESSION_FLAG_PS`
Persistent Session.

`BFD_MSG_SESSION_FLAG_AD`
Session in Admin Down state.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_SUCCESS when the function succeeds

bfd_echo_interval_set

This function sets the BFD echo mode transmission interval for all single-hop sessions on an interface.

Syntax

```
s_int32_t
bfd_echo_interval_set (u_int32_t vr_id, s_int32_t proc_id,
                       char *name, u_int32_t echo_tx)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
	Process ID for the BFD instance <0–65535>.
`name`	Interface name.
`echo_tx`	Echo transmission interval.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR_PMIRROR_ENABLED when pmirror is enabled on an interface

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_echo_interval_unset

This function resets the BFD echo mode transmission interval to its default value for all sessions on an interface.

Syntax

```
s_int32_t
bfd_echo_interval_unset (u_int32_t vr_id, s_int32_t proc_id, char *name)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`name`	Interface name.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_interval_set

This function sets the minimum BFD transmission and reception intervals, and the Hello multiplier value for all sessions on an interface.

Syntax

```
s_int32_t
bfd_proto_interval_set (u_int32_t vr_id, s_int32_t proc_id, u_int32_t ifindex,
                        char *name, u_int32_t min_tx, u_int32_t min_rx,
                        u_int32_t multiplier)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.
`name`	Interface name.
`min_tx`	Minimum transmission interval.
`min_rx`	Minimum reception interval.
`multiplier`	Hello multiplier value.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_interval_unset

This function resets the BFD transmission and reception intervals, and the Hello multiplier value to their default values for all sessions on an interface.

Syntax

```
s_int32_t
bfd_proto_interval_unset (u_int32_t vr_id, s_int32_t proc_id, uint32_t ifindex)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_multihop_interval_set

This function sets the minimum BFD transmission and reception intervals, and the Hello multiplier value for all multihop sessions on an IPv4 interface.

Syntax

```
s_int32_t
bfd_proto_multihop_interval_set (u_int32_t vr_id, s_int32_t proc_id,
                                 s_int32_t ifindex, struct pal_in4_addr ipv4,
                                 u_int32_t min_tx, u_int32_t min_rx,
                                 u_int32_t multiplier)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index; this value is generally 0 for all multihop sessions not bound to an interface.
`ipv4`	IPv4 address.
`min_tx`	Minimum transmission interval.
`min_rx`	Minimum reception interval.
`multiplier`	Hello multiplier value.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_MH_NOT_FOUND when the multihop instance is not found or allocated

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_multihop_interval_unset

This function resets the BFD transmission and reception intervals, and the Hello multiplier value to their default values for all multihop sessions on an interface.

Syntax

```
s_int32_t
bfd_proto_multihop_interval_unset (u_int32_t vr_id, s_int32_t proc_id,
                                   u_int32_t ifindex, struct pal_in4_addr ipv4)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index; this value is generally 0 for all multihop sessions not bound to an interface.
`ipv4`	IPv4 address.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_MH_NOT_FOUND when the multihop instance is not found or allocated

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_protol_multihop_ipv6_interval_set

This function sets the minimum BFD transmission and reception intervals, and the Hello multiplier value for all multihop sessions on an IPv6 interface.

Syntax

```
s_int32_t
bfd_proto_multihop_ipv6_interval_set (u_int32_t vr_id, s_int32_t proc_id,
                                      u_int32_t ifindex,
                                      struct pal_in6_addr ipv6,
                                      u_int32_t min_tx, u_int32_t min_rx,
                                      u_int32_t multiplier)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index; this value is generally 0 for all multihop sessions not bound to an interface.
`ipv6`	IPv6 address.
`min_tx`	Minimum transmission interval.
`min_rx`	Minimum reception interval.
`multiplier`	Hello multiplier value.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_MH_NOT_FOUND when the multihop instance is not found or allocated

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_protol_multihop_ipv6_interval_unset

This function resets the BFD transmission and reception intervals, and the Hello multiplier value to their default values for all multihop sessions on an IPv6 interface.

Syntax

```
s_int32_t
bfd_proto_multihop_ipv6_interval_unset (u_int32_t vr_id, s_int32_t proc_id,
                                        u_int32_t ifindex,
                                        struct pal_in6_addr ipv6)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index; this value is generally 0 for all multihop sessions not bound to an interface.
`ipv6`	IPv6 address.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_MH_NOT_FOUND when the multihop instance is not found or allocated

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_echo_mode_set

This function sets BFD to echo mode for all interfaces in a BFD process.

Syntax

```
s_int32_t
bfd_echo_mode_set (u_int32_t vr_id, s_int32_t proc_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.

Output Parameters

None

Return Values

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_echo_mode_unset

This function removes the BFD echo mode setting for all interfaces in a BFD process.

Syntax

```
s_int32_t
bfd_echo_mode_unset (u_int32_t vr_id, s_int32_t proc_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.

Output Parameters

None

Return Values

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_slow_timer_set

This function sets a BFD slow timer for all interfaces in a BFD process.

Syntax

```
s_int32_t
bfd_proto_slow_timer_set (u_int32_t vr_id, s_int32_t proc_id, u_int32_t val)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`val`	Timer value in milliseconds.

Output Parameters

None

Return Values

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_slow_timer_unset

This function resets the BFD slow timer to the default value (BFD_SLOW_TIME_DEFAULT) for all interfaces in the BFD process.

Syntax

```
s_int32_t
bfd_proto_slow_timer_unset (u_int32_t vr_id, s_int32_t proc_id)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.

Output Parameters

None

Return Values

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_slow_timer_set_interface

This function sets the slow timer value for a particular interface. This Hello interval is used when the session is not up.

Syntax

```
s_int32_t
bfd_proto_slow_timer_set_interface (u_int32_t vr_id, s_int32_t proc_id,
                                    char *ifname, u_int32_t val)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifname`	Interface name.
`val`	Timer value in milliseconds.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_SET_ERROR_PMIRROR_ENABLED when pmirror is enabled on an interface

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

bfd_proto_slow_timer_unset_interface

This function unsets the slow timer to the default value (1 sec) for a particular interface.

Syntax

```
s_int32_t
bfd_proto_slow_timer_unset_interface (u_int32_t vr_id, s_int32_t proc_id,
                                      u_int32_t ifindex)
```

Input Parameters

`vr_id`	Virtual router ID. Default value is 0. Pass 0 for a non-VR implementation.
`proc_id`	Process ID for the BFD instance <0–65535>.
`ifindex`	Interface index.

Output Parameters

None

Return Values

BFD_API_SET_ERR_VR_NOT_EXIST when the VR does not exist

BFD_API_INSTANCE_NOT_FOUND when the instance is not found

BFD_API_IF_NOT_FOUND when the interface is not found

BFD_API_SET_ERROR when a generic error occurs

BFD_API_SET_SUCCESS when the function succeeds

CHAPTER 14 BFD MIB Support

This chapter describes the API for the BFD management information base (MIB).

Overview

BFD managed objects are accessed through a virtual information store using Simple Network Management Protocol (SNMP). It describes managed objects to configure and/or monitor BFD for both single-hop and multi-hop sessions. ZebOS-XP modules act as the sub-agent that communicates with the master agent using AgentX protocol. Master Agent ideally runs in the agent. When SNMP Manager makes a request to the agent, the agent sends a corresponding request to the ZebOS-XP sub-agent through AgentX protocol. Similarly, the response sent by the sub-agent to the master agent is sent back to the manager by the agent.

For BFD MIB objects, when an SNMP request arrives from the SNMP master agent to BFD (which registers the BFD MIB with the Master Agent), the BFD module finds the corresponding data structures and accesses or updates the request. For a Get request, the object value is retrieved and sent to the master agent. For a Set request, the corresponding data structure is modified.

For notifications defined in the BFD MIB, which are generated from the BFD module, a Trap message is sent to the Master agent. The master agent forwards this message to the SNMP Trap manager registered with the agent. A limitation is imposed on the number of traps at regular intervals, in order to avoid flooding of traffic between both the SNMP master agent – sub-agent (ZebOS-XP) and the SNMP master agent – SNMP manager. The limit is set to a maximum of 5 traps per minute.

BFD Session Table

The session table (bfdSessTable) is used to identify a BFD session between a pair of nodes. The index for this table is bfdSessIndex, which is used to uniquely identify the BFD sessions. This table has the objects of types, read-only and read-create.

Attribute	Syntax	Access	Functions
bfdSessTable	BfdSessIndexTC	not-accessible	bfd_sess_lookup_by_index
bfdSessTable	-	-	bfd_sess_lookup_next_by_index
bfdSessTable/ bfdSessNotificationsEnable	TruthValue	read-write	bfd_notification_set
bfdSessVersionNumber	Unsigned32 (0..7)	read-create	bfd_api_set_sess_version_no
bfdSessType	INTEGER { singleHop(1), multiHopTotallyArbitraryPaths(2), multiHopOutOfBandSignaling(3), multiHopUnidirectionalLinks(4) }		bfd_api_get_sess_type

bfdSessMultihopUniLinkMode	Integer {none(1), active(2), passive(3)}	read-only	bfd_api_get_sess_mh_unlnk_mode
bfdSessDiscriminator	Unsigned32 (0 | 1..4294967295)	read-only	bfd_api_get_sess_disc
bfdSessRemoteDiscr	Unsigned32 (0 | 1..4294967295)	read-only	bfd_api_get_sess_rmte_disc
bfdSessDestinationUdpPort	InetPortNumber	read-create	bfd_api_get_sess_dest_udp_port
bfdSessSourceUdpPort	InetPortNumber	read-create	bfd_api_get_sess_src_udp_port
bfdSessEchoSourceUdpPort	InetPortNumber	read-create	bfd_api_get_sess_echo_src_udp_port
bfdSessAdminStatus	INTEGER { stop(1), start(2)}	read-create	bfd_api_get_sess_admin_status
bfdSessState	INTEGER { adminDown(1), down(2), init(3), up(4), failing(5) }	read-only	bfd_api_get_sess_state
bfdSessRemoteHeardFlag	TruthValue	read-only	bfd_api_get_sess_rmte_heard_flag
bfdSessDiag	BfdDiag	read-only	bfd_api_get_sess_diag
bfdSessOperMode	INTEGER {asyncModeWEchoFun(1), asynchModeWOEchoFun(2), demandModeWEchoFunction(3), demandModeWOEchoFunction(4) }	read-create	bfd_api_get_sess_oper_mode
bfdSessDemandModeDesiredFlag	TruthValue	read-create	bfd_api_get_sess_dmnd_mode_dsrd_flag
bfdSessControlPlaneIndepFlag	TruthValue	read-create	bfd_api_get_sess_cntrlplane_indep_flag
bfdSessInterface	InterfaceIndexOrZero	read-create	bfd_api_get_sess_interface
bfdSessAddrType	InterfaceIndexOrZero	read-create	bfd_api_get_sess_addr_type
bfdSessAddr	InetAddressType	read-create	bfd_api_get_sess_addr
bfdSessGTSM	TruthValue	read-create	bfd_api_get_sess_gtsm
bfdSessGTSMTTL	Unsigned32 (0..255)	read-create	bfd_api_get_sess_gtsm_ttl
bfdSesDesiredMinTxInterval	BfdInterval	read-create	bfd_api_get_sess_dsrd_min_tx_intvl
bfdSessRequiredMinRxInterval	BfdInterval	read-create	bfd_api_get_sess_req_min_rx_intvl

bfdSessReqMinEchoRxInterval	BfdInterval	read-create	bfd_api_get_sess_req_min_echo_rx_intvl
bfdSessDetectMult	Unsigned32	read-create	bfd_api_get_sess_detectmult
bfdSessNegotiatedInterval	BfdInterval	read-only	bfd_api_get_sess_neg_intvl
bfdSessNegotiatedEchoInterval	BfdInterval	read-only	bfd_api_get_sess_neg_echo_intvl
bfdSessNegotiatedDetectMult	Unsigned32	read-only	bfd_api_get_sess_neg_detect_mult
bfdSessAuthPresFlag	TruthValue	read-create	bfd_api_get_sess_auth_pres_flag
bfdSessAuthenticationType	INTEGER {reserved(0), simplePassword(1), keyedMD5(2), meticulousKeyedMD5(3), keyedSHA1(4), meticulousKeyedSHA1(5) }	read-create	bfd_api_get_sess_auth_type
bfdSessAuthenticationKeyID	Integer32 (-1 | 0..255)	read-create	bfd_api_get_sess_auth_key_id
bfdSessAuthenticationKey	OCTET STRING (SIZE (0..252))	read-create	bfd_api_get_sess_auth_key
bfdSessStorType	StorageType	read-create	bfd_api_get_sess_stor_type
bfdSessRowStatus	RowStatus	read-create	bfd_api_get_sess_row_status
bfdSessPerfPktIn	Counter32	read-only	bfd_api_get_perf_pkt_in
bfdSessPerfPktOut	Counter32	read-only	bfd_api_get_perf_pkt_out
bfdSessUpTime	TimeStamp	read-only	bfd_api_get_sess_up_time
bfdSessPerfLastCommLostDiag	BfdDiag	read-only	bfd_api_get_perf_lastcomm_lost_diag
bfdSessPerfSessUpCount	Counter32	read-only	bfd_api_get_perf_sess_up_count
bfdSessPerfDiscTime	TimeStamp	read-only	bfd_api_get_perf_disc_time
bfdSessPerfPktInHC	Counter64	read-only	bfd_api_get_perf_pkt_in_hc
bfdSessPerfPktOutHC	Counter64	read-only	bfd_api_get_perf_pkt_out_hc

APIs

The following subsection list the APIs for BFD.

bfd_sess_lookup_by_index

This function loops through the BFD sessions for SNMP Get request based on the session index.

Syntax

```
struct bfd_session *
bfd_sess_lookup_by_index (struct bfd *bfd, u_int32_t sess_index)
```

Input Parameters

`*bfd`	A pointer to the BFD structure
`sess_index`	Session index

Output Parameters

None

Return Values

BFD session structure from the session list based on the index if it exists; otherwise, NULL is returned.

bfd_sess_lookup_next_by_index

This function loops through the BFD sessions for SNMP Getnext or Walk request based on the session index.

Syntax

```
struct bfd_session *
bfd_sess_lookup_next_by_index (struct bfd *bfd, u_int32_t sess_index)
```

Input Parameters

`bfd`	A pointer to the BFD structure
`sess_index`	Session index

Output Parameters

None

Return Values

Next BFD session structure from the list, based on the index next to the given index if it exists; otherwise, NULL is returned.

bfd_notification_set

This function sets the notification traps.

Syntax

```
s_int32_t
bfd_notification_set (u_int32_t vr_id, u_int32_t set)
```

Input Parameters

`vr_id`	Virtual router ID
`set`	Value to be set

Output Parameters

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_version_no

This function is used to set the session version for an SNMP set request.

Syntax

```
s_int32_t
bfd_api_set_sess_version_no (struct bfd *bfd, u_int32_t index,
                             u_int32_t intval, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameters

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_version

This function gets a session version.

Syntax

```
s_int32_t
bfd_api_get_sess_version (struct bfd *bfd,u_int32_t index,
                          u_int32_t *ret, bool_t snmp)
```

Input Parameters

`*bfd`	A pointer to the BFD structure
`index`	Index
`snmp`	Enable SNMP

Output Parameters

`ret`	Return value

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

BFD_VERSION_SUPP

bfd_api_set_sess_addr_type

This function is used to set the session address type.

Syntax

```
s_int32_t
bfd_api_set_sess_addr_type (struct bfd *bfd, u_int32_t index,
                            u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameters

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_type

This function is used to get the session type for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_type (struct bfd *bfd,u_int32_t index,
                       u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session type when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_mh_unlnk_mode

This function is used to get the session multihop UNI link mode for SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_mh_unlnk_mode (struct bfd *bfd, u_int32_t index,
                                u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the multihop UNI link mode of a session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_disc

This function is used to get the session discriminator for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_disc (struct bfd *bfd,u_int32_t index,
                       u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session discriminator when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_rmte_disc

This function is used to get the session remote discriminator for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_rmte_disc (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session remote discriminator when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_dest_udp_port

This function is used to get the session destination UDP port for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_dest_udp_port (struct bfd *bfd,u_int32_t index,
                                u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the destination UDP port number of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_src_udp_port

This function is used to set the session source UDP port for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_src_udp_port (struct bfd *bfd,u_int32_t index,
                               u_int32_t intval, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_src_udp_port

This function is used to get the session source UDP port for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_src_udp_port(struct bfd *bfd,u_int32_t index,
                              u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the source UDP port number of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_echo_src_udp_port

This function is used to set the session echo source UDP port for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_echo_src_udp_port (struct bfd *bfd,u_int32_t index,
                                    u_int32_t intval, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_echo_src_udp_port

This function is used to get the session echo source UDP port for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_echo_src_udp_port (struct bfd *bfd,u_int32_t index,
                                    u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the echo source UDP port number of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_admin_status

This function is used to set the session administration status for an SNMP set request.

Syntax

```
s_int32_t
bfd_api_set_sess_admin_status (struct bfd *bfd,u_int32_t index,
                               u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_admin_status

This function is used to get the session administration status for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_admin_status (struct bfd *bfd,u_int32_t index,
                               u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session administration status when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_state

This function is used to get the session state for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_state (struct bfd *bfd,u_int32_t index,
                        u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns session state when the function succeeds or 0 when it fails: BFD_API_SESS_ST_AD_DWN BFD_API_SESS_ST_DWN BFD_API_SESS_ST_UP BFD_API_SESS_ST_INIT

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_rmte_heard_flag

This function is used to get the session remote heard flag for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_rmte_heard_flag (struct bfd *bfd,u_int32_t index,
                                  u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the remote heard flag of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_diag

This function is used to get the session diagram for a SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_diag (struct bfd *bfd,u_int32_t index,
                       u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session diagram when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_oper_mode

This function is used to get the session OperMode for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_oper_mode (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the operational mode of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_dmnd_mode_dsrd_flag

This function is used to set the session demand mode desired flag for an SNMP set request.

Syntax

```
s_int32_t
bfd_api_set_sess_dmnd_mode_dsrd_flag (struct bfd *bfd,u_int32_t index,
                                      u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_dmnd_mode_dsrd_flag

This function is used to get the session demand mode desired flag for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_dmnd_mode_dsrd_flag (struct bfd *bfd,u_int32_t index,
                                      u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the demand mode desired flag of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_cntrlplane_indep_flag

This function is used to get the session control plane independent flag for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_cntrlplane_indep_flag (struct bfd *bfd,u_int32_t index,
                                        u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the control plane independent flag of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_interface

This function is used to set the session interface for an SNMP set request.

Syntax

```
s_int32_t
bfd_api_set_sess_interface (struct bfd *bfd,u_int32_t index,
                            u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_interface

This function is used to get the session interface for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_interface (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session interface when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_addr_type

This function is used to set the session address type for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_addr_type (struct bfd *bfd, u_int32_t index,
                            u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_addr_type

This function is used to get the session address type for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_addr_type (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session address type when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_addr6

This function sets an IPv6 session address for an SNMP set request

Syntax

```
s_int32_t
bfd_api_set_sess_addr6 (struct bfd *bfd,u_int32_t index,
                        struct pal_in6_addr *addr,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`addr`	Enable SNMP
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_addr

This function is used to get the session address for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_addr (struct bfd *bfd,u_int32_t index,
                       char **ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session address when the function succeeds or NULL when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_gtsm

This function is used to set the session GTSM for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_gtsm (struct bfd *bfd,u_int32_t index,
                       u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_gtsm

This function is used to get the session GTSM for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_gtsm (struct bfd *bfd,u_int32_t index,
                       u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session address type when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_gtsm_ttl

This function is used to set the session GTSM TTL for an SNMP set request.

Syntax

```
s_int32_t
bfd_api_set_sess_gtsm_ttl (struct bfd *bfd,u_int32_t index,
                           u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_gtsm_ttl

This function is used to get the session GTSM TTL for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_gtsm_ttl (struct bfd *bfd,u_int32_t index,
                           u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session GTSM TTL when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_dsrd_min_tx_intvl

This function is used to set the session desired minimum transmission interval for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_dsrd_min_tx_intvl (struct bfd *bfd,u_int32_t index,
                                    u_int32_t min_tx,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`min_tx`	Minimum BFD transmission in milliseconds
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_dsrd_min_tx_intvl

This function is used to get the session desired minimum transmission interval for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_dsrd_min_tx_intvl (struct bfd *bfd,u_int32_t index,
                                    u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the desired minimum transmission interval of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_req_min_rx_intvl

This function is used to set the session required minimum receive interval for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_req_min_rx_intvl (struct bfd *bfd,u_int32_t index,
                                   u_int32_t min_rx,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`min_rx`	Minimum BFD reception in milliseconds
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_req_min_rx_intvl

This function is used to get the session required minimum receive interval for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_req_min_rx_intvl (struct bfd *bfd,u_int32_t index,
                                   u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the required minimum receive interval of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_req_min_echo_rx_intvl

This function is used to set the session required minimum echo receive interval for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_req_min_echo_rx_intvl (struct bfd *bfd,u_int32_t index,
                                        u_int32_t echo_int,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`echo_int`	Echo interval
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_req_min_echo_rx_intvl

This function is used to get the session required minimum echo receive interval for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_req_min_echo_rx_intvl (struct bfd *bfd,u_int32_t index,
                                        u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the required minimum echo receive interval of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_detect_mult

This function is used to set the session detect multiplier for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_detect_mult (struct bfd *bfd,u_int32_t index,
                              u_int32_t mult,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`mult`	Detection multiplier
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_detectmult

This function is used to get the session detect multiplier for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_detectmult (struct bfd *bfd,u_int32_t index,
                             u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the detect multiple value of the session when the function succeeds or 0 when it fails.

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_neg_intvl

This function is used to get the session negotiated interval for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_neg_intvl (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the negotiated interval of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_neg_echo_intvl

This function is used to get the session negotiated echo interval for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_neg_echo_intvl (struct bfd *bfd,u_int32_t index,
                                 u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the negotiated echo interval of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_neg_detect_mult

This function is used to get the session negotiated detect multiplier for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_neg_detect_mult (struct bfd *bfd,u_int32_t index,
                                  u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the negotiated detect multiple value of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_auth_pres_flag

This function is used to get the session authentication preserve flag for an SNMP Get request.

Syntax

```
u_int32_t
bfd_api_get_sess_auth_pres_flag (struct bfd *bfd,u_int32_t index,
                                 u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the authentication preserve flag of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_auth_type

This function is used to get the session authentication type for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_auth_type (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session authentication type when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_auth_key_id

This function is used to get the session authentication key ID for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_auth_key_id (struct bfd *bfd,u_int32_t index,
                              u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the authentication key ID of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_auth_key

This function is used to get the session authentication key for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_auth_key (struct bfd *bfd,u_int32_t index,
                           char **ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the authentication key of the session when the function succeeds or NULL when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_stor_type

This function is used to set the session storage type for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_stor_type (struct bfd *bfd,u_int32_t index,
                            u_int32_t intval,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`intval`	Value to be set
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_stor_type

This function is used to get the session storage type for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_stor_type (struct bfd *bfd,u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session storage type when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_set_sess_row_status

This function is used to set the session row status for an SNMP Set request.

Syntax

```
s_int32_t
bfd_api_set_sess_row_status (struct bfd_master *bm, u_int32_t index,
                             u_int32_t val,  bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`val`	Status value
`snmp`	Enable SNMP

Output Parameter

None

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_row_status

This function is used to get the session row status for an SNMP Get request.

Syntax

```
s_int32_t
bfd_api_get_sess_row_status (struct bfd *bfd,u_int32_t index,
                             u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Session index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the session row status when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_perf_pkt_in

This function gets the number of performance packet received.

Syntax

```
u_int32_t
bfd_api_get_perf_pkt_in (struct bfd *bfd, u_int32_t index,
                         u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the number performance packet received when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_perf_pkt_out

This function gets the number of performance packet forwarded.

Syntax

```
s_int32_t
bfd_api_get_perf_pkt_out (struct bfd *bfd, u_int32_t index,
                          u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the number of performance packets forwarded when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_sess_up_time

This function gets the session up time parameter.

Syntax

```
s_int32_t
bfd_api_get_sess_up_time (struct bfd *bfd, u_int32_t index,
                          u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the up time parameter of the session when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_perf_lastcomm_lost_diag

This function gets the performance last communication lost diagnostic.

Syntax

```
s_int32_t
bfd_api_get_perf_lastcomm_lost_diag (struct bfd *bfd, u_int32_t index,
                                     u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the performance last communication lost diagnostic when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_perf_sess_up_count

This function gets the performance session up count parameter.

Syntax

```
s_int32_t
bfd_api_get_perf_sess_up_count (struct bfd *bfd, u_int32_t index,
                                u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the performance session up count parameter when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_perf_disc_time

This function gets the performance discriminator time parameter.

Syntax

```
s_int32_t
bfd_api_get_perf_disc_time (struct bfd *bfd, u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the performance discriminator time parameter when the function succeeds or 0 when it fails

Return Values

None

bfd_api_get_perf_pkt_in_hc

This function gets the number of high-capacity performance packets received.

Syntax

```
s_int32_t
bfd_api_get_perf_pkt_in_hc (struct bfd *bfd, u_int32_t index,
                            u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the number of high-capacity performance packets received when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_api_get_perf_pkt_out_hc

This function gets the number of high-capacity performance packet forwarded.

Syntax

```
s_int32_t
bfd_api_get_perf_pkt_out_hc (struct bfd *bfd, u_int32_t index,
                             u_int32_t *ret, bool_t snmp)
```

Input Parameters

`bfd`	A pointer to BFD structure
`index`	Packet index
`snmp`	Enable SNMP

Output Parameter

`ret`	Returns the number of high-capacity performance packets forwarded when the function succeeds or 0 when it fails

Return Values

BFD_FAILURE when the function fails

BFD_SUCCESS when the function succeeds

bfd_sess_lookup_by_disc_map_index

This function loops through the BFD sessions for SNMP Get request based on the session local discriminator.

Syntax

```
struct bfd_session *
bfd_sess_lookup_by_disc_map_index (u_int32_t loc_disc,
                                   u_int8_t exact, struct bfd *bfd)
```

Input Parameters

`loc_disc`	Local discriminator
`exact`	Exact value
`bfd`	A pointer to BFD structure

Output Parameters

None

Return Values

BFD session structure from the session tree based on the local discriminator if it exists; otherwise, NULL is returned.

Traps and Scalar Objects

Traps and scalar objects for the BFS session state are defined in this section.

Notifications

BFD Session Up Notification (bfdSessUp)

This notification is generated when the bfdSessState object for one or more contiguous entries in bfdSessTable are about to enter the up (4) state from some other state.

BFD Session Down or Session Admin Down Notification (bfdSessDown)

This notification is generated when the bfdSessState object for one or more contiguous entries in bfdSessTable are about to enter the down (2) or adminDown (1) state from some other state.

Scalar Variables

BFD Admin Status (bfdAdminStatus)

The global administrative status of BFD in this router. The value “enabled” (1) denotes that the BFD Process is active on at least one interface; “disabled” (2) disables it on all interfaces.

BFD Session Notification Enable (bfdSessNotificationsEnable)

If this object is set to true (1), then it enables the emission of bfdSessUp and bfdSessDown notifications; otherwise, these notifications are not emitted.

The MIB is only for IP-based BFD sessions and BFD objects available for both Single-hop and Multi-hop sessions.

Index

A

B

C

F

H

I

M

N

O

P

R

S

T