ZebOS-XP®
Network Platform

Version 1.4
Extended Performance

RSVP-TE 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 Resource Reservation—Traffic Engineering (RSVP-TE).

Audience

This guide is intended for developers who write code to customize and extend RSVP-TE.

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, *Internal Structures*
- Chapter 3, *Messages*
- Chapter 4, *Label Switched Router Roles*
- Chapter 5, *Constrained Shortest Path First*
- Chapter 6, *Network Services Module*
- Chapter 7, *Label Pool Manager*
- Chapter 8, *Quality of Service*
- Chapter 9, *MPLS Forwarder*
- Chapter 10, *Differentiated Services*
- Chapter 11, *Refresh Reduction*
- Chapter 12, *Fast Reroute*
- Chapter 13, *Command API*

- Chapter 14, *Management Information Base API*
- Chapter 15, *Point-to-Multipoint Services*

Related Documents

The following guides are related to this guide:

- *RSVP-TE Command Reference*
- *Multi-Protocol Label Switching Command Reference*
- *Multi-Protocol Label Switching Configuration Guide*
- *Multi-Protocol Label Switching Developer Guide*
- *Multi-Protocol Label Switching Software Forwarder Developer 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

Resource ReSerVation Protocol (RSVP) is a signaling protocol that supports explicit routing capabilities. To do this, a simple Explicit Route (ER) object is incorporated into the RSVP PATH messages. The object encapsulates a sequence of hops, which constitute the explicitly-routed path. Using ER object, the paths taken by the label-switched RSVP-MPLS flows can be predetermined without conventional IP routing. An explicitly-routed path can be administratively determined, or computed based on the Constrained Shortest Path First (CSPF) algorithm and policy requirements dictated by the operator through a trunk node.

One useful application of explicit routing is Traffic Engineering (TE). Using explicitly routed LSPs, an ingress node can control the path through which traffic flows from itself, through the MPLS network, to the egress node. Explicit routing is therefore useful for the optimization of network resources and an increase in the quality of traffic-oriented performance.

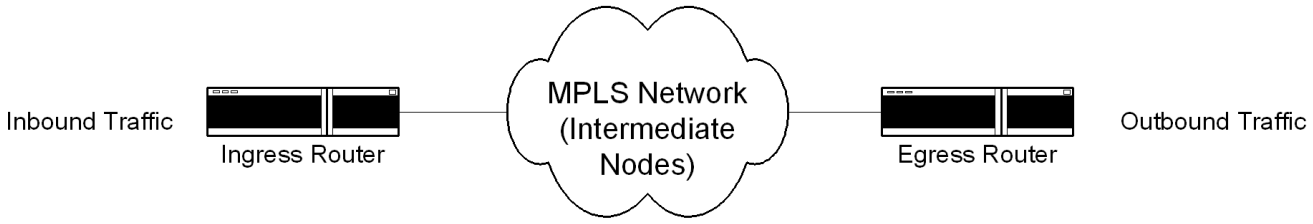

Figure 1-1: Overview of MPLS Network

RSVP-TE Architecture

RSVP-TE is a signaling protocol that supports explicit routing capabilities to establish Label-Switched Paths (LSPs) in an MPLS (Multi-Protocol Label Switching) network.

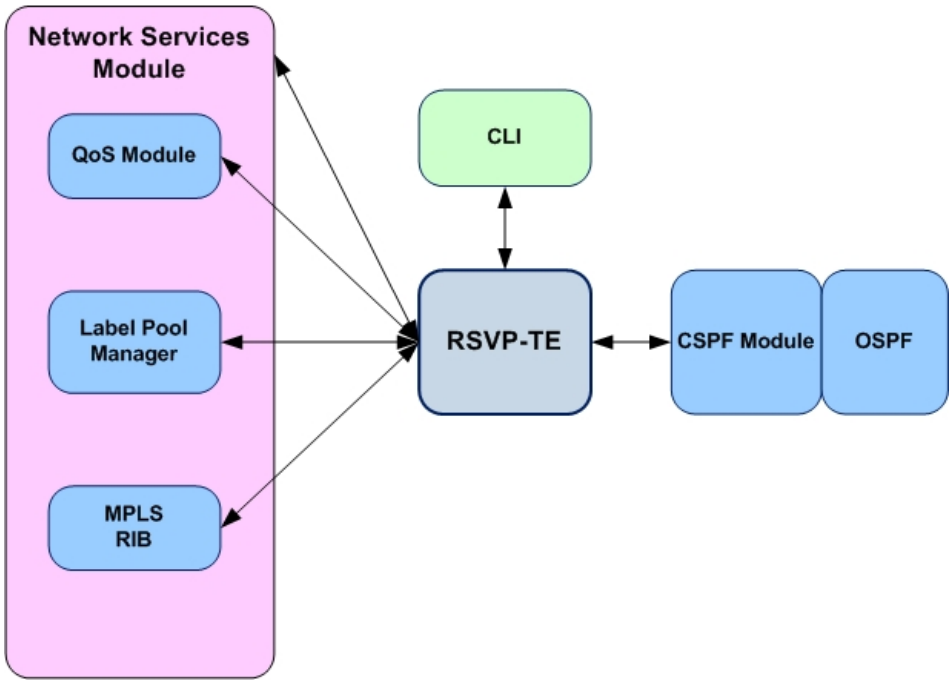

Figure 1-2: RSVP-TE Interaction with NSM

Features

ZebOS-XP RSVP-TE:

- creates explicitly routed paths, which might not agree with the route suggested by the IGP (OSPF, RIP) used. Explicitly routed LSPs, by definition, do not follow the paths suggested by IGPs.
- queries CSPF for a complete, end-to-end, explicit route based on constraints configure by the operator using the RSVP CLI.
- preempts existing LSPs with lower priority in order to accommodate an LSP with a higher priority. This functionality is part of the Quality of Services (QoS) module of the NSM.
- performs make-before-break type re-routing of tunnels. Make-before-break is the creation of a new LSP before the old one is torn down.
- exchanges Hello messages to make node failures easier to detect. This means when there is no exchange between routers, then other node is assumed dead or offline, except when the peer is known to not support Hellos.
- supports integrated services through the exchange of Integrated Services (IntSrv) objects in RSVP messages.
- provides statistical information of RSVP messages exchanged.

In addition, ZebOS-XP RSVP-TE can be used with BGP to create MPLS/BGP Virtual Private Networks (VPN), and with the Label Distribution Protocol (LDP) to create Layer-2 Virtual Circuits.

CHAPTER 2 Internal Structures

These are the basic structures used in ZebOS-XP RSVP-TE:

Read-Write Thread

ZebOS-XP RSVP-TE uses a separate socket to send and receive messages between RSVP-TE peers. Each thread is bound to a unique socket, so that each socket is treated as a unidirectional mode of exchange.

Write Queue

ZebOS-XP RSVP-TE maintains a write queue filled with messages to write to a given LSR. The write thread is turned on when there is something to write to the network, otherwise it is canceled. For each message to write to the network, a write-queue-node is created, which contains:

- Size of the data to be sent out
- Data stream
- Outgoing interface
- Destination prefix
- Router alert option to set in the IP header
- TTL to set in the IP header

Once data is sent, its corresponding write-queue-node is deleted.

Trunks

RSVP maintains a table of all the trunks configured on a router. All the sessions for which a router is either egress or transit are also stored in this table. A trunk is used for all the sessions between a given set of routers having the same tunnel identifier and the same extended tunnel identifier. A trunk is identified by a unique tuple <tunnel-id, egress, extended-tunnel-id> where tunnel-id is the tunnel identifier assigned by the ingress, egress is the IP address of the egress router and extended-tunnel-id is the extended tunnel identifier assigned by the ingress router.

Paths

RSVP maintains a list of all configured paths. New paths are added to the tail of the list.

There are two identifiers for a path: a 16-bit identifier generated by the system and a string provided by the operator.

A path specifies the list of strict or loose hops to be taken by an explicitly-routed LSP:

- If a hop is loose, the hop must be taken by the LSP during setup. However, there is no constraint on the next hop.
- If a hop is strict, the hop must be taken by the LSP during setup, and it should have followed a strict hop, if there was a strict hop in the path right before this hop—and it should be followed by a strict hop —if there is a strict hop in the path immediately following this hop.

Explicit and Implicit Neighbors

Neighbors or peers for an LSR or an LER may be defined in two ways:

- Explicity, via a command:

```
!
router rsvp
 neighbor 1.2.3.4
!
```

- Implicitly, via receipt of an RSVP message from a directly-connected LSR, which leads to an explicit peer definition.

You can choose to exchange Hello messages among neighbors using the `hello-receipt` command.

Extended Timer

The extended timer handles MPLS RSVP-TE thread operations.

The number of timer slots is 60, with a timer granularity of 1 second, creating a minute wheel as shown in Figure 2-1. Each timer-slot holds a list of sorted timer threads. The indexing of the timer slots plays an important role in adding, executing, and deleting the timer thread.

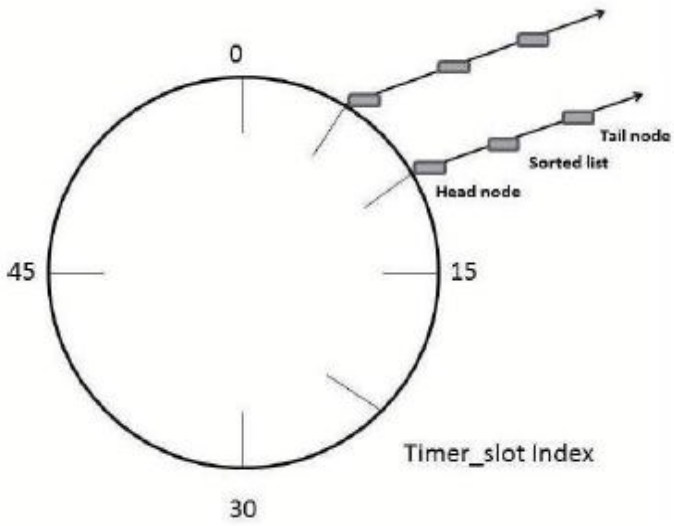

Figure 2-1: Extended Timer Index

Operations

Addition

- The timer thread occurs on an index on the timer slots. The calculation of this index depends on the current index on the timer-slots and the value of the timer which needs to be added.
- This timer thread is then placed in the sorted list on that index.

Execution

- The `thread_fetch` function runs until there is at least one thread available for I/O or timer, after which, it quits the loop.
- Each time `thread_fetch` is called, only few indices (between the current time and the previous time a timer thread was executed) will be chosen to loop for expired threads. This current time will be backed up as the previous time for the next execution. Similarly, the next index will become the new current index for both the timer addition and for the next execution.

Deletion

Index is backed up during thread addition. This aids in quickly locating the index and deleting the thread.

CHAPTER 3 Messages

Message Content

Every RSVP message is divided into two main sections:

- A header
- A body

An RSVP packet header contains the following:

- RSVP Version (Current version is 1)
- Flags (unused)
- Checksum for the RSVP message
- TTL value for the RSVP message
- Length of the RSVP message

An RSVP message body must contain at least one RSVP message.

Note: IP Infusion's RSVP implementation can send and receive one only RSVP message per packet; bundling is not supported.

Message Types

The following messages are used by ZebOS-XP RSVP-TE:

- Path (PATH) Message
- Reservation (RESV) Message
- Path Tear (PTEAR) Message
- Reservation Tear (RTEAR) Message
- Path Error (PERR) Message
- Reservation Error (RERR) Message
- Reservation Confirmation (RCONF) Message
- Hello Message

Path (PATH) Message

A Path message is initiated from an ingress node. Its purpose is to identify the nodes to be traversed by an LSP. The Path message may be directed to the egress, if traditional routing is being used, or to the first hop in the explicit route object (ERO), if an explicit path is being configured. Each intermediate node that receives a Path message creates a Path State Block (PSB), and passes on the Path messages to the next hop. Upon receipt of the Path message at the egress node, a Reservation message is sent to the ingress. A Path message is sent out periodically by the ingress, egress and intermediate nodes.

Reservation (RESV) Message

An RESV message is initiated from an egress node. It is sent hop-by-hop towards the ingress. Upon receipt of a RESV message, each node reserves resources, provides an MPLS Label to be used by incoming MPLS packets, and passes on a RESV message to the next hop. RESV messages are sent out periodically by the egress and intermediate nodes.

Path Tear (PTEAR) Message

A PTEAR message may be initiated by an ingress or an intermediate node. This message is sent towards the egress in the same way that the Path message was originally sent. The PTEAR deletes the PSB and frees all resources reserved.

Reservation Tear (RTEAR) Message

An RTEAR message may be initiated by the egress or an intermediate node. This message is sent towards the ingress in the same way that the RESV message was originally sent. Upon receipt of an RTEAR, a node frees up resources but does not delete the PSB. The timer to send out periodic Path messages remains active.

Path Error (PERR) Message

A PERR message may be initiated by an egress or an intermediate node. This message denotes an error in the received PATH message and deletes the PSB on all intermediate nodes that it encounters on its way to the ingress. Upon receipt of a PERR on the ingress, the LSP setup may be retried if retry parameters are configured.

Reservation Error (RERR) Message

An RERR message may be initiated by an ingress or an intermediate node. This message denotes an error in reservation and releases all reserved resources on nodes that it encounters on its way to the egress.

Reservation Confirmation (RCONF) Message

An RCONF message is sent by the ingress, to the egress, in reply to a Reservation Message containing an RCONF object. An egress may choose to add an RCONF object to a Reservation message in order to get confirmation of a complete LSP setup. ZebOS-XP RSVP does not add RCONF objects to Reservation messages but is fully capable of handling RCONF requests from other vendors.

Hello Message

Hello messages are optional messages supported by ZebOS-XP RSVP-TE. The primary aim of Hello messages is to detect link failures without having to wait for message timeouts. Once two peers are identified as Hello-exchange capable, an Hello message is exchanged between the two. If no Hello message is received from a node, which was previously identified as being Hello-exchange-capable - for a specified period of time, it is assumed that the peer's link has gone down and all PSBs shared with the peer are reset.

CHAPTER 4 Label Switched Router Roles

An Label Switched Router (LSR) performs one of the following three roles for a Label Switched Path (an LSP):

- Ingress Node
- Intermediate Node
- Egress Node

Ingress Node

The ingress is a node where an RSVP trunk is defined. Refer to the *Internal Structures* chapter for more information on trunk attributes. Upon successful definition of a trunk, a Path State Block (PSB) is created using the attributes provided by the trunk. At the ingress node, the following messages are sent/received:

- If CSPF is available and enabled for the LSP, a query is sent to the CSPF server for a complete ERO list. If the reply is favorable, a PATH message is sent out to the first hop in the ERO list. If the reply is a notification or an error, an appropriate error is generated and the retry timer may be initiated, if it is configured.
- If CSPF is not available, a PATH message is sent out to the first hop in the existing ERO list, or to the egress (if there is no ERO list) and a primary path is configured.
- Upon sending a PATH message — state changes to PATH SENT — an ingress node may receive either a Path Error (PERR), Reservation (RESV), or a Reservation Tear (RTEAR) message, and manage it as described in the following sections.

Path Error Message

Receipt of a Path Error message prompts the session to be broken down, the state transitions to NON EXISTENT and the retry timer comes into play, if one was configured.

Reservation Message

If state is PATH SENT, receipt of a Resv Error message completes the end-to-end LSP setup and the transitions to OPERATIONAL. Reservation for the LSP takes place, if possible, and a new FTN entry is added to the MPLS Forwarder. If state is already OPERATIONAL, the reservation timeout timer is reset each time only.

Reservation Tear Message

A Resv Tear message can only be received if the state is OPERATIONAL. Receipt of this message changes the state to PATH SENT, and all reserved resource(s) are released, along with the FTN entry initially created, thereby successfully breaking the LSP.

If no messages are received, timeout occurs after specified (keep-multiplier * refresh) seconds.

Intermediate Node

The following message exchange is related to an intermediate node:

- Receipt of a PATH message causes the creation of a Path State Block for the LSP being requested, if one does not already exist. If the received PATH message is acceptable, an updated PATH message is forwarded downstream, and the state is changed to PATH SENT. If the received PATH message is unacceptable, a PERR message is sent upstream, and the PSB is deleted.
- Receipt of an acceptable RESV message causes the reservation of a resource for the LSP on the outgoing link, installation of a label in the MPLS Forwarder for the LSP, transmission of a RESV message to the upstream peer, and change of state to OPERATIONAL. Upon receipt of an invalid/unacceptable RESV message, a RERR message is sent to the downstream peer, and state is changed to PATH SENT, if it was previously OPERATIONAL.
- Receipt of a PTEAR/PERR causes the transmission of a PTEAR/PERR to the upstream peer, and the PSB to be deleted.
- Receipt of an RERR causes the transmission of an RERR to the downstream peer, release of all reserved resources if any, and change of state to PATH SENT, if it was previously OPERATIONAL.
- RTEAR: This message can only be received if the state is OPERATIONAL Receipt of this message will change state to PATH SENT, and all reserved resource(s) will be released, along with the FTN entry initially created, thereby successfully breaking the LSP. Also, an RTEAR message will be sent to the upstream peer.

Egress Node

The following message exchange is related to an intermediate egress correct node:

- Receipt of a PATH message causes the creation of a Path State Block for the LSP being requested, if one doesn't already exist. If the received PATH message is acceptable, a RESV message is forwarded upstream, and the state is changed to OPERATIONAL. If the received PATH message is unacceptable, a PERR message is sent upstream, and the PSB is deleted.
- Receipt of a RERR/PTEAR causes the PSB to be deleted. This, in turn, removes all entries from the MPLS Forwarder, and release any resource(s) that were reserved, hence breaking the LSP.
- An RTEAR message can only be received if the state is OPERATIONAL. Receipt of this message changes the state to PATH SENT, and all reserved resources are released along with the FTN entry initially created, thereby successfully breaking the LSP. Additionally, an RTEAR message is sent to the upstream peer.
- If a resource on the egress is preempted, or is released by some form of intervention, an RTEAR message is sent upstream, and the PSB on the egress is deleted.

CHAPTER 5 Constrained Shortest Path First

RSVP-TE communicates with the Constrained Shortest Path First (CSPF) module via an IPC to obtain an optimum Explicit Route (ER) based on specified constraints. It uses the Traffic Engineering Database (TED) and the pre-existing Label Switched Paths (LSP). The resulting ER is used to set up the traffic-engineered LSP. For more aboiut CSPF, see the *Open Shortest Path First Developer Guide.*

Interfaces

RSVP-TE interfaces are defined in the sections that follow.

Initialization

RSVP-TE establishes connection with the CSPF module and initializes internal structures.

```
int  rsvp_cspf_init (void);
```

LSP Query

RSVP-TE sends this message to CSPF to request a route computation based on a set of specified constraints. CSPF returns a Route message when the calculation can be completed. It returns a Notification message when an error occurs. This message is sent under the following conditions:

- Computing a new route
- Re-computing an old route (possibly with changed attributes)

```
int rsvp_cspf_lsp_query (struct rsvp_session *session, u_char req_type);
```

LSP Established

RSVP-TE sends this message to CSPF to confirm the setup of a signaled LSP. This message should be sent for LSPs which are setup using CSPF (for computation), as well as, LSPs which are manually setup (without using CSPF)

```
int rsvp_cspf_lsp_established (struct rsvp_session *session);
```

LSP Deletion Request

RSVP-TE sends this message to CSPF to delete a given LSP identified by LSP ID.

```
int rsvp_cspf_lsp_delete (struct rsvp_session *session);
```

Disconnect

RSVP-TE disconnects with CSPF module and cleans up its internal structures.

```
void rsvp_cspf_deinit (u_char free_sessions);
```

It is installed into the forwarder in the event of a failure on protected LSP.

CHAPTER 6 Network Services Module

The Network Services Module (NSM) sends unsolicited messages to, or receives unsolicited messages from, the QoS (quality of service) module. These messages and their structures are discussed below.

Message Sent by the NSM

The NSM should be able to enable interfaces for QoS. It should therefore send an interface initiate message to the QoS module.

Message Received by the NSM

Maximum reservable bandwidth is reported by the QoS module to the NSM. The maximum reservable bandwidth is a number that corresponds to the total maximum bandwidth reserved by all LSPs using a particular interface. For example, for a 100 MBPS link, a maximum reservable bandwidth of 200 MBPS dictates that the total bandwidth reserved by all LSPs using this link cannot exceed 200 MBPS, and any single reservation may not exceed 100 MBPS. The QoS module may choose to ignore the maximum reservable bandwidth, or it may rely only on the bandwidth attribute, which is an intrinsic property of the interface.

Upon successful reservation for an LSP, the available bandwidth per-priority-level may change. This change is made known to the NSM via an unsolicited message so that the NSM may pass on this information to CSPF. Information about available bandwidth per-priority-level is stored in the QoS module. The QoS module provides a global view of this information.

Interaction of NSM with Signaling Protocols

The NSM interacts minimally with the signaling protocols. It passes on the following information to the signaling protocol so that the application can determine whether requesting a reservation is feasible.

- Bandwidth per interface allows the signaling protocol to decide whether the requested bandwidth is actually possible.
- Maximum reservable bandwidth
- Bandwidth per priority level

CHAPTER 7 Label Pool Manager

To manage the RSVP labels, RSVP-TE communicates with the Label Pool Manager using an API provided by the NSM. For more about the Label Pool Manager API, see tthe *Network Services Module Developer Guide*.

Initialization

RSVP-TE initializes a label object and establishes connection to the Label Pool Manager.

```
struct rsvp_label *rsvp_label_init (u_int32_t out_label, u_int16_t label_space,
                                    struct prefix *fec,
                                    session_role_t session_role,
                                    struct rsvp_label*old_label,u_char php_in_use);
```

Receive a Label Request

RSVP-TE receives a label request and processes it. This function should be invoked at the egress only.

```
int rsvp_label_req_recv (struct rsvp_session *,
                         struct rsvp_label_request *, int *, u_char *);
```

Receive a Label Object Request

RSVP-TE receives a label request and processes it. This function should be invoked at ingress and interim nodes only.

```
int
int rsvp_label_obj_recv (struct rsvp_session *, struct rsvp_label_obj *, int *);
```

Delete a Label Object

RSVP-TE deletes a label object.

```
void rsvp_label_delete (struct rsvp_label *, struct rsvp_session *, bool_t, bool_t);
```

CHAPTER 8 Quality of Service

The Quality of Service (QoS) module interacts with the NSM, the signaling protocol (RSVP-TE) and the MPLS Forwarder.

Overview

The QoS module acts as a server and requires its clients to register for specific types of services. These services are broken down into the following types:

- Bandwidth Data (for use by the NSM)
- Traffic Data (for use by RSVP-TE)
- Statistical information per interface (used by signaling protocols)

Each client registers for one or more of these services, using a mask to identify the service requirements. Traffic data can be broken down into two parts, in case CR-LDP does not require all the attributes that RSVP-TE requires.

RSVP-TE communicates with the QoS module, which currently resides inside the NSM. The QoS module is the server and the RSVP-TE daemon is the client. The QoS client is provided with a set of APIs to probe for, reserve and release resources. These APIs are part of the QoS client library in `lib/qos_client.[ch]`).

QoS Wrappers

RSVP-TE uses the following QoS-specific wrappers in order to carry out the various resource procedures:

Table 8-1: QoS-Specific Wrappers and Usage

`rsvp_qos_init`	Initializes connection to the QoS server
`rsvp_qos_deinit`	De-initializes the QoS server
`rsvp_qos_probe`	Fnds if the requested bandwidth is available at the priority level specified. If the probe fails, a Path Error message is sent upstream for intermediate nodes, or no Path message is sent out for ingress nodes.
`rsvp_qos_reserve`	Reserves bandwidth for an LSP. If no bandwidth can be reserved, a Reservation Error is sent towards the egress. If pre-existing LSP needs to be preempted in order to make space for the new LSP, preemption messages are sent to the clients.
`rsvp_qos_release`	Releases a reserved resource

In addition to the wrappers defined above, the QoS server may send an unsolicited preemption message to the protocol, which is handled by RSVP-TE using the `rsvp_qos_preempt_handler` wrapper.

Interaction with Signaling Protocol

The signaling protocol, in this case RSVP-TE, can either be the initiator of a request, or the recipient of updates.

Signaling Protocol as Initiator

When a new LSP is defined on an ingress node, the following information is required:

- Availability of the requested bandwidth on the outgoing interface
- High-enough setup priority of the requested LSP for it to be created, and the list of resources that need to be pre-empted

This is a QoS module attribute, since a given signaling protocol might not be aware of resources reserved by another signaling protocol on the same box. The QoS module should maintain a global view of available bandwidth per priority level (0 to 7) so that it responds correctly to queries from multiple signaling protocols.

Given the bandwidth requested, and the peak information provided by the operator, the QoS module suggests bucket attributes that suit the specified traffic flow. Refer to the `qos_common.h` file. These messages return an appropriate status message to the signaling protocol:

Therefore, an “exploratory” message is provided to the signaling protocol, which allows it to get traffic shaping information and determine out if the LSP in question can be set up or not, without actually reserving bandwidth. This message should be asynchronous.

A reservation message is also provided for the signaling protocol to actually reserve the requested bandwidth.

QoS Module as Initiator

The QoS module sends the following unsolicited messages to the signaling protocol:

- Changes in interface statistical information to all clients registered for the statistical information service. This way RSVP, for example, sends out a PATH message without waiting for the regular PATH-send interval.
- Bucket-specific information that was initially suggested by the QoS module may change, given the load on a given interface. This information is immediately sent to the client registered for traffic specialization service. The signaling protocol may choose to ignore it, or may wish to update the LSP, as per local policy. Bucket-change messages should be identified by a QoS resource.
- Keeping in mind that the QoS module maintains a global bandwidth per -priority-level table, an LSP might need to be pre-empted to accommodate another LSP at a higher priority level. The QoS module is therefore able to send pre-emption messages to its clients, and the clients are able to respond with an appropriate status.

CHAPTER 9 MPLS Forwarder

RSVP-TE uses the MPLS APIs provided by the NSM to populate ILM and FTN entries in the MPLS RIB. The MPLS RIB is maintained by the NSM. The NSM may then choose to install ILM or FTN entry of RSVP-TE in the Forwarder, if required.

Add ILM Entry

RSVP-TE adds the specified ILM entry to the ILM table.

```
int rsvp_mpls_ilm_ipv4_label_install (struct rsvp_session *session, u_int32_t iif_ix,
                                      u_int32_t in_label);
```

Add FTN Entry

RSVP-TE adds the specified FTN entry to the FTN table.

```
int rsvp_mpls_ftn_ipv4_label_install (struct rsvp_session *session, struct prefix *p);
```

Remove ILM Entry

RSVP-TE removes the specified ILM entry from the ILM table.

```
int rsvp_mpls_ilm_ipv4_label_remove (struct rsvp_session *session, u_int32_t iif_ix,
                                     u_int32_t in_label);
```

Remove FTN Entry

RSVP-TE removes the specified FTN entry from the FTN table.

```
int rsvp_mpls_ftn_ipv4_label_install (struct rsvp_session *session, struct prefix *p);
```

CHAPTER 10 Differentiated Services

The Differentiated Services (DiffServ) framework provides quality of service in a network by applying packet-classification rules at the edge of the network. All packets belonging to a specific traffic class receive related forwarding treatment at each node in a DiffServ-compliant network. This forwarding behavior is called Per-Hop-Behavior (PHB). In MPLS networks, the packets are classified setting by EXP bits in the MPLS label appended to the packet. The other way to classify packets is by using a different label for each supported service type.

Terminology

Table 10-1: DiffServ terminology

BA	Behavior Aggregate	A collection of packets with the same DiffServ codepoint crossing a link in a particular direction
DSCP	Differentiated Services Code Point	A specific value of the DSCP portion of the DiffServ field, used to select a PHB.
PHB	Per Hop Behavior	The externally observable forwarding behavior (Per Hop Behavior) applied at a DiffServ-compliant node to a DiffServ behavior aggregate
OA	Ordered Aggregate	The set of Behavior Aggregates that share an ordering constraint
PSC	PHB Scheduling Class	The set of one or more PHB(s) that are applied to the Behavior Aggregate(s) belonging to a given OA. For example, AF1x is a PSC comprising the AF11, AF12 and AF13 PHBs. EF is an example of PSC comprising a single PHB, the EF PHB
E-LSP	EXP-Inferred-PSC LSP	Label for which the DiffServ behavior is inferred from the EXP bits in the MPLS header
L-LSP	Label-Only-Inferred-PSC LSP	Label for which the DiffServ behavior is inferred from the label in the MPLS header

DiffServ LSP Types

DiffServ relies on two types of LSPs: E-LSP and L-LSP.

E-LSP

An E-LSP is used to support one or more OAs. Such LSPs can support up to eight BAs of a given FEC, regardless of how many OAs these BAs span. With E-LSPs, the EXP field of the MPLS Shim Header is used by the LSR (label switch router) to determine the PHB to apply to the packet, including both the PSC and the drop preference.

The mapping from the EXP field to the PHB for a given LSP, is either explicitly signaled at label set-up (signaled E-LSP) or relies on a pre-configured mapping (pre-configured E-LSP).

L-LSP

An L-LSP is a separate LSP which can be established for a single <FEC, OA> pair. With L-LSPs, the PSC is explicitly signaled at the time of label establishment, so that subsequently the LSR can infer exclusively from the label value the PSC to apply to a labeled packet. When the Shim Header is used, the Drop Precedence to be applied by the LSR to the labeled packet is conveyed inside the labeled packet MPLS Shim Header using the EXP field.

RSVP DiffServ Object

To support the DiffServ feature, the RSVP DiffServ Object is part of the RSVP PATH. This object is used to carry the EXP-to-PHB mapping for signaled E-LSP or the PSC value for L-LSP. For detailed information about this object, refer to section 5.2 of RFC 3270.

DiffServ Configuration Commands

For details about DiffServ commands, see the *RSVP-TE Command Reference*.

Backward Compatibility

The `override-diffserv` command supports backward-compatibility, When you give the `override-diffserv` command, all received PATH messages that do not have an RSVP DiffServ Object are considered as requests to set up a non-DiffServ LSP. When this flag is reset using the `no override-diffserv` command, all received PATH messages that do not have RSVP DIffServ Object are considered as requests to set up a preconfigured E-LSP.

CHAPTER 11 Refresh Reduction

This chapter contains a technical overview of the Refresh Reduction mechanisms.

Overview

RSVP-TE is a “soft-state” protocol and all RSVP-TE speakers are designed to detect changes in the state by examining refresh messages that are periodically exchanged between all RSVP-TE speakers. Such refresh chatter is costly for the following reasons:

- Typically, each RSVP-TE message formulates its own IP packet. In an environment where thousands of Label Switched Paths (LSPs) are used, this can lead to a large number of interrupts in the control plane processor.
- If any change in state is to be propagated to RSVP-TE speaking neighbors, a new PATH or RESV message is generated. To detect changes in state, the receiving node must parse and compare all received data with previously received information for the corresponding LSP. This processing is costly, especially when there is no state change to propagate.

IETF RFC 2961 addresses both of these issues by providing means to minimize chatter, and by indirectly introducing ways to indicate whether the state for an LSP has been modified.

Terminology

Trigger Messages

Trigger messages include messages advertising a new state, a route change that alters a reservation path, or a modification to an existing RSVP session or reservation. Trigger messages provide new or updated state information and may be used to remove specific LSP state information.

Refresh Messages

Refresh Messages represent previously-advertised states and contain exactly the same objects and same information as a previously-transmitted message, and are sent over the same path. Only PATH and RESV messages can be refresh messages.

Refresh Reduction Mechanisms

To support Refresh Reduction, an RSVP-TE speaker must be able to identify all LSPs for which the node needs to send messages to a common neighbor. The neighbor data structure should therefore provide information about all LSPs - upstream and downstream - that share the same neighbor.

To limit the number of messages exchanged between neighbors, all refresh messages sent to a neighbor that supports Refresh Reduction, are queued on a per-neighbor basis (refer to RFC2961 for further details on the Srefresh and Ack queues). Messages in a queue are sent out in the following cases:

1. When the number of messages in the queue are enough to generate one IP packet.
2. When the first message added to the queue is queued for a set period of time (3 seconds).
3. When a trigger message needs to be sent to the same neighbor. In this case, the trigger message causes some or all messages in the queue to be bundled along with the trigger message.

Note: Messages sent to neighbors not supporting Refresh Reduction are not queued. Such messages are sent out in the default manner.

There are two basic Refresh Reduction mechanisms to minimize RSVP-TE chatter: Message Bundling and Summary Refresh. ZebOS-XP RSVP-TE supports the Summary Refresh mechanism.

Summary Refresh

Summary Refresh (Srefresh) messages allow refresh messages to be summarized with 32-bit message identifiers. For each PATH or RESV message that need to be summarily refreshed, the sending node adds a Message ID object in the message. In ZebOS-XP RSVP-TE, sending an ACK message is optional. If acknowledgement of a Message ID object is requested, a Message ID ACK message may be sent to the requesting neighbor.

Each Message ID object carries a 24-bit epoch value and a 32-bit identifier that corresponds to the state to refresh. An epoch value denotes a steady state for an RSVP speaker; a change denotes that the transmitting node has restarted. Epochs are associated with outgoing interfaces, and are regenerated in the following events:

- When the Message ID counter used to summarily refresh PATH/RESV states needs to be wrapped around
- When the RSVP process is started or restarted
- When the interface is activated or reactivated

If the epoch value in a received message for a given PATH or RESV state is not the same as the one received earlier for the same PATH or RESV state, the new message must be treated as a trigger message.

If the message-identifier value in a received message for a PATH or RESV state is greater than the one received earlier for the same PATH or RESV state, the newly received message must be treated as a trigger message.

Combined with the source IP address of the Srefresh message and the 32-bit identifiers in the Message ID list received in the Srefresh message, the receiving node can figure out the state block to which this information corresponds. In the event that no corresponding state block is found, a Message ID NACK message must be sent to the appropriate neighbor.

While Summary Refresh messaging is in use, refresh RESV and PATH messages need not be exchanged at default intervals. To ensure ample interval, during Refresh Reduction, the refresh-time for affected state blocks is increased by multiplying it by 1000.

Epoch Values

Epoch values must be generated for each interface enabled for RSVP signaling.

Message ID values

Each RSVP-capable interface is required to support a running counter for Message ID values. All messages sent from an interface pick up the next available index from the associated Message ID counter of the interface.

If the counter needs to be rolled back (if `oxffffffff` has already been used as a Message ID), the epoch value for the corresponding interface is regenerated, and the implementation can then start reusing all available Message ID values.

Retransmit Queues

Trigger messages carrying Message ID objects may be sent out with an acknowledgement-desired flag set. The acknowledgement-desired flag may only be set for all messages that are generated due to transitions through the appropriate finite state machine (FSM), and not for messages generated due to refresh interface time-outs. For all refresh interface time-out messages, a copy is maintained in a retransmit queue. Each message has an associated timer. IP Infusion recommends a default timeout of 3 seconds. If a corresponding acknowledgement is received for the Message ID, the corresponding node in the retransmit queue is removed. In the event that no acknowledgement is received before the time-out expires, a new message is sent to the appropriate neighbor.

Each RSVP-enabled interface maintains its own retransmit queue. Messages are added to the queues based on the outgoing interface to use to transmit them.

Buffering of PATH/RESV States for Srefresh

For Summary Refresh (SRefresh) messages, each neighbor maintains one or more message identifier lists (depending on the number of PATH/RESV states being refreshed) for timely transmission to neighbors. The lists are maintained for periodic refreshes, and are only reprocessed if either a state has been deleted, or a new state needs to be refreshed in the list. A change in an epoch value also causes reprocessing of lists.

Size Considerations for an RSVP Message

ZebOS-XP RSVP-TE assures that no RSVP message exceeds the Maximum Transmission Unit (MTU) for the outgoing interface. Although this restriction is advisable for generic RSVP messages, summary-refresh support may be hampered due to these restrictions.

Known Stack Limitation

Most operating systems do not support the fragmentation of locally-generated raw packets. The maximum size for SRefresh messages is the MTU for the link, which may lead to a bottleneck. Ideally, SRefresh messages should be able to occupy a complete IP packet. If the operating system supports fragmentation, minor changes to the code allows the Summary Refresh message-size to satisfy a complete IP packet.

Note: Support for Bundle Messaging is also dependant on the operating system’s support for fragmentation.

CHAPTER 12 Fast Reroute

This chapter contains an architectural overview of ZebOS-XP RSVP-TE Fast Reroute (FRR).

Topics in this chapter include:

Overview

Fast reroute is used in MPLS networks to enable redirection of traffic onto predesignated backup Label Switched Paths (LSPs) in the event of a failure on a primary or protected LSP. The backup LSP tunnel is signaled before failure of the protected LSP to minimize loss of data and achieve fast traffic redirection. Protected LSPs can be backed up using a one-to-one protection mechanism or a many-to-one (facility backup) mechanism. ZebOS-XP FRR supports both mechanisms.

Fast Reroute Terminology

Fast Reroute . Fast Reroute (FRR) refers to redirection of traffic onto backup LSPs in the event of a failure. .

One-to-one Backup. The one-to-one backup method creates one backup (or detour) LSP for each protected LSP at . each potential point of local repair (PLR).

Facility Backup. In the facility backup method, a backup LSP (bypass tunnel) is created to protect one or more . protected LSPs that traverse the PLR, the resource being protected, and the Merge Point in that order.

Local Repair. Local Repair refers to the techniques used to repair protected LSPs when a link or node along the path. fails.

Point of Local Repair. A Point of Local Repair (PLR) is an LSR that initiates FRR procedures in the event of a failure. on the protected LSP. This LSR is also the head end of the designated backup LSP being used for redirected traffic.

Protected LSP. An LSP is termed as Protected at a given node in the network if it has at least one backup tunnel . originating at that node.

Detour LSP. A detour LSP is used for backing up a given protected LSP in the one-to-one backup scenario. .

Bypass Tunnel. A bypass tunnel is used to protect a set of LSPs passing over a common facility. .

Merge Point. A Merge Point (MP) is a node where one or more backup LSPs rejoin the path of the protected LSP . downstream of the potential failure. This also acts as the egress for bypass tunnels.

RSVP-TE Fast Reroute

In this section, the terms “LSP” and “session” are used interchangeably.

Trunk or Session Storage

A trunk is identified by the tuple [Tunnel ID, Egress IP address, Extended Tunnel ID]. All LSPs that belong to the same trunk must contain the same trunk identifiers. All detour LSPs belong to the same trunk as the associated protected LSP, which is useful for LSP merging. To support this requirement, the trunk table key uses the tuple defined above as the primary key in the table.

Each trunk contains a list of associated sessions (LSPs). These sessions are identified by the tuple [Ingress IP address, LSPID]. A primary and its backup might have the same session tuple (for detour LSPs using the Path identification method). In these cases, a flag (primary or backup) is used for LSP identification. A primary and its backup must have the same LSPID. The list keeps all LSPs with the same LSPID together followed by LSPs with different LSP IDs.

Trunk Name

Only Ingress trunks have corresponding names. Each session instance stores session names locally instead of relying on the trunk for storage. This is necessary because different sessions in the same trunk may have different name attributes.

Trunk Router Role

The router role attribute is part of the RSVP-TE session definition.

RSVP Message Parser

The FRR object should be forwarded unchanged by routers which do not understand this object. This capability is part of the RSVP parser.

Backup Path Signaling

A backup path is required for each LSP being protected at a given LSR. This path must avoid the resource being protected. Fast switchover can only be achieved if the backup path is pre-signaled to avoid signaling delays.

In the one-to-one mechanism, the backup LSP should be signaled before failure on the primary LSP to avoid loss of data.

In the facility backup method, the PLR determines a label to indicate to the MP that packets received with that label should be switched along the protected LSP. This is done without explicitly signaling the backup path if the MP uses a label space global to that LSR. When MPs use per-interface label spaces, the PLR sends Path messages (for each protected LSP using a bypass tunnel) via that bypass tunnel prior to the failure in order to discover the appropriate MP label.

Local Repair Techniques

Two Local Repair techniques, the one-to-one backup and the facility backup are used to redirect traffic in case of failure. ZebOS-XP FRR supports both Local Repair techniques.

This section contains information about the following subjects:

One-to-One Backup Method

In the one-to-one backup local repair technique, a separate backup LSP (detour) is signaled for each protected LSP.

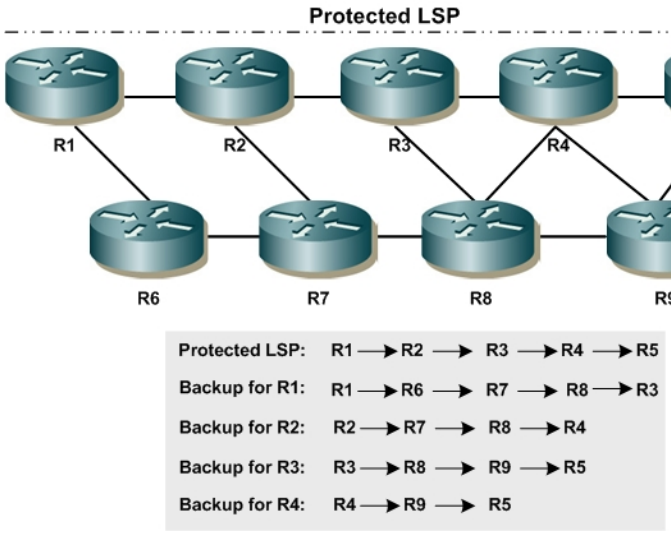

Figure 12-1: One-to-One Backup Topology

The topology above depicts an example of the one-to-one backup mechanism. In this example, the protected LSP runs from R1 to R5. To provide protection, R2 creates a partial backup LSP that merges with the protected LSP at R4. This partial backup LSP [R2->R7->R8->R4] is called a detour .

Detours are signaled at each node along the path of the LSP where protection is desired. To fully protect an LSP, a detour is signaled at each node along the path of the protected LSP, except at the egress. Each such node is termed as a Point of Local Repair (PLR).

Detour LSPs must have the same egress node as the protected LSP. It may be merged with the protected LSP downstream of the potential failure to reduce signaling overhead. Detours may have different attributes (for example, bandwidth, priority, etc.) than the protected LSP.

A Detour LSP can be identified using one of the following methods.

Sender Template Method

In this method, a detour shares the RSVP Session object and LSPID with the protected LSP and changes the ingress IP address in the RSVP PATH message. According to the RSVP resource sharing rules, this LSP can be merged with the protected LSP as they have same session object.

Path Specific Method

In this method, a new RSVP object (DETOUR) is added to the PATH message to differentiate it from the protected LSP's path messages. Since, a detour has the same session object as the protected LSP, it might share common network resources.

One-to-One Backup Scenario

This section provides an overview of the events and actions taken by the modules that support FRR procedures when using the one-to-one backup scenario.

An LSP is configured on the ingress node using the related Fast Reroute commands. The fast-reroute protection command is used to designate that the one-to-one backup FRR mechanism is desired.

The operator should set the desired attributes of the LSP using the attribute commands related to the one-to-one protection method. See the *RSVP-TE Command Reference* for one-to-one protection command details. If no attributes are assigned, a backup LSP is created with the same attributes as the protected LSP. These attributes are signaled for detour LSPs and also signaled in the protected LSP using the Fast Reroute (FRR) object.

Two sessions are created, one for protected and one for detour LSPs, at the ingress node, and both instances are stored in the same trunk node.

The PATH messages for the protected LSP contain the FRR object that is used by downstream nodes, along the path of the protected LSP, to signal local detours. Detour LSPs may merge with the protected LSP downstream of the failure.

If the FRR object changes, then the PLR should propagate the new attributes when signaling the detour LSP.

The downstream nodes wait for a RESV message for the protected LSP before signaling detour LSPs. Waiting is required for the PLR to gather information about nodes and links being traversed by protected LSP to use for backup path computation.

Once the detour LSP is operational, the Ingress adds a backup FTN entry to the primary FTN entry created by the protected LSP.

Transit nodes add a backup ILM entry for the ILM entry created by the protected LSP.

In the event of primary LSP failure, the corresponding secondary (FTN or ILM) entries are installed to the MPLS forwarder by NSM. The PLR sends an error message to the ingress node which may then recompute a new optimal path for the primary LSP.

Facility Backup Method

In the facility backup local repair technique, a backup LSP (bypass tunnel) is signaled for all protected LSPs.

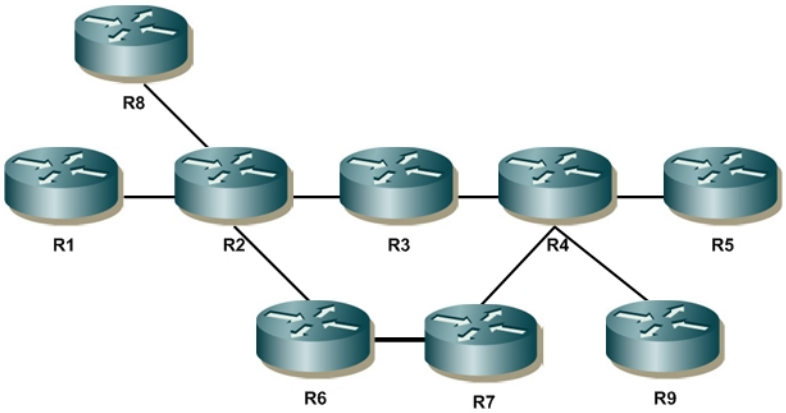

Figure 12-2: Facility Backup Topology

The topology above depicts an example of the facility backup mechanism. In this example, the bypass tunnel R6->R7 is the backup LSP created to protect link R2->R3 and node R3. The same bypass tunnel can also be used to protect LSP from any of R1, R2, or R8 to any of R4, R5, or R9.

When a failure occurs along a protected LSP, the PLR redirects traffic into the appropriate bypass tunnel. For instance, if link [R2->R3] fails in this example, R2 will switch traffic received from R1 on the protected LSP onto link [R2->R6]. The label will be switched for one which will be understood by R4 to indicate the protected LSP, and the bypass tunnel's label will then be pushed onto the label stack of the redirected packets.

If penultimate-hop-popping is used, the merge point in this example, R4, will receive the redirected packet with a label indicating the protected LSP that the packet is to follow. If penultimate-hop-popping is not used, R4 will pop the bypass tunnel's label and examine the label underneath to determine the protected LSP that the packet is to follow. When R2 is using the bypass tunnel for protected LSP 1, the traffic takes the path [R1->R2->R6->R7->R4->R5]; the bypass tunnel is the connection between R2 and R4.

Facility Backup Scenario

This section contains an overview of the events and actions taken by the modules that support FRR procedures when using the facility backup scenario.

An LSP is configured on the ingress node using the related Fast Reroute commands. Use the `fast-reroute protection` command to designate that the facility backup FRR mechanism is desired.

The operator should specify the desired attributes of the LSP using the attribute commands related to the one-to-one protection method. See the *RSVP-TE Command Reference* for facility backup protection command details. If no attributes are assigned, a backup LSP is created with the same attributes as the protected LSP. These attributes are signaled for detour LSPs and also in the protected LSP using the Fast Reroute (FRR) object.

The selection of a bypass tunnel for a protected LSP is performed by the PLR when the LSP is first set up. The PATH messages for the protected LSP contain the FRR object that is used by downstream nodes, along the path of the protected LSP, to signal the bypass tunnel. Bypass tunnels should merge with the protected LSP downstream of the failure. If the FRR object changes, then the PLR propagates the new attributes when signaling the bypass LSP.

Label Switched Path Failure

When a failure occurs along a protected LSP, the PLR redirects traffic onto the local detour LSP. An LSP is designated as failed or down when any of the following conditions are met

- Outgoing interface goes down
- Error notification from Layer-2 is received
- Nexthop node goes down
- Incoming interface on the nexthop node goes down
- RSVP refresh message times out
- RSVP Error messages are received (for example, PathTear and ResvTear)

Referring to the one-to-one backup example topology in Figure 12-1, if link [R2->R3] fails, R2 switches traffic received from R1 onto the protected LSP along link [R2->R7]. R2 uses the label it received when creating the detour. When R4 receives traffic with the label provided for the detour of R2, R4 switches that traffic onto link [R4-R5] using the label received from R5 for the protected LSP. The depth of the label stack does not increase as a result of taking the detour. While R2 is using its detour, traffic takes the path [R1->R2->R7->R8->R4->R5].

Referring to the facility backup example topology in Figure 12-2, if link [R2->R3] fails, R2 switches traffic received from R1 onto the protected LSP along link [R2->R6], and starts sending control traffic for the protected LSP onto the bypass tunnel. R2 uses the label it received when creating the bypass tunnel. The label is switched for one which is understood by R4 to indicate the protected LSP, and the bypass tunnel's label is then onto the label-stack of the redirected packets. If penultimate-hop-popping is used, the merge point in this example, R4 receives the redirected packet with a label indicating the protected LSP that the packet is to follow. If penultimate-hop-popping is not used, R4 pops the bypass tunnel's label and examines the label underneath to determine the protected LSP that the packet is to follow; in this case, R4->R5. While R2 is using the bypass tunnel, traffic takes the path [R1->R2->R6->R7->R4->R5].

Revertive Behavior

ZebOS-XP FRR restores a failed traffic engineered LSP to a full working path using the Global Revertive Mode. In this mode, the head-end of the failed LSP is responsible for recomputing and signaling a new path.

RSVP-TE Protocol Extensions

RSVP-TE protocol is extended to support the following:

- Fast Reroute Object
- Detour Object
- Session Attribute Flags
- RRO IPV4/IPV6 Subobject Flags

Fast Reroute Object

The properties of backup LSPs can be communicated by either signaling a Fast Reroute Object in RSVP PATH messages or by setting a Local Protection Desired flag in the session attribute object in the PATH message.

When the Fast Reroute Object is signaled, it provides information about the desired bandwidth, priorities and other attributes for the backup LSP. While signaling an LSP for which protection is desired, the ingress node inserts this object in the PATH messages to indicate the desired properties of backup LSP to downstream nodes. See RSVP-TE Fast Reroute for details. When the Local Protection Desired flag is set, a PLR node creates backup LSPs with the

same attributes as the protected LSP. In case the Fast Reroute Object and the Local Protection Desired flag both are set, the attributes set in the Fast Reroute Object take precedence over the attributes of the protected LSP.

Note: The Fast Reroute Object must be passed unchanged to all downstream nodes.

Detour Object

The detour object is carried in RSVP PATH messages while signaling backup LSPs in one-to-one backup scenarios. This object acts as an identifier for backup LSPs. It carries the IP address of the head end of the detour LSP (PLR) and also the IP address of the immediate downstream node of the PLR, which should be avoided.

Session Attribute Flags

The session attribute flag contains with the following values.

Bandwidth Protection Desired

This flag indicates to the PLRs along the protected LSP path that a bandwidth guarantee is desired for backup LSPs. The actual bandwidth value is deduced from FRR object or the value present in the PATH message of the protected LSP.

Node Protection Desired

This flag indicates to PLRs that the backup LSP should bypass at least the next node of the initiating PLR.

RRO IPV4/IPV6 Subobject Flags

The following flags are defined for the RRO IPV4/IPV6 subobject.

Bandwidth Protection

This flag is set by a PLR when a backup LSP with the desired bandwidth guarantee is operational.

Node Protection

This flag is set by a PLR when a backup LSP is operational which bypasses the next node along the protected LSP path.

Local Protection in Use

This flag is set by a PLR when a protected LSP is using local detour for forwarding data.

Network Services Module Extensions

NSM functionality extensions to support FRR are detailed below.

ILM Table

ILM table needs to support concept of associated backup NHLFE entries for a given incoming label entry. This backup entry contains NHLFE information (such as an outgoing label and nexthop interface) signaled by the backup LSP. This data is installed in the forwarder to use in the event of the failure of a protected LSP.

Fast Reroute Commands

The commands required to configure backup LSP support for RSVP signaled LSPs are in the *RSVP-TE Command Reference.* This guide contains all of the commands necessary to:

- Create a new RSVP Bypass Tunnel or to modify an existing RSVP Bypass Tunnel
- Display detailed information for all bypass tunnels
- Create a Fast Reroute backup and to specify the protection type (one-to-one or facility backup)
- Assign attributes to the FRR backup LSP for either the one-to-one or facility backup protection method
- Set the backup LSP identification method for one-to-one protection
- Assign attributes to the FRR backup LSP for facility backup protection

CHAPTER 13 Command API

The functions in this chapter apply to both IPv4 and IPv6.

rsvp_api_path_get

This call creates new RSVP path with the specified name, if not already found.

Syntax

```
struct rsvp_path *rsvp_api_path_get (const char *name)
```

Input Parameters

`name`	RSVP path name

Output Parameters

None

Return Value

The path

rsvp_api_path_delete

This call deletes the specified RSVP path.

Syntax

```
struct rsvp_path *rsvp_api_path_delete (const char *name)
```

Input Parameters

`name`	RSVP path name

Output Parameters

None

Return Value

RSVP_SUCCESS when the RSVP path is successfully deleted

RSVP_FAILURE when the RSVP path is not successfully deleted

rsvp_api_path_hop_add

This call adds a hop to the ERO path.

Syntax

```
int
rsvp_api_path_hop_add (struct rsvp_path *path, struct rsvp_path_ero *ero,
```

```
                        struct prefix *p, u_char is_strict, u_char add_to_top)
```

Input Parameters

`path`	RSVP path
`ero`	ERO path
`p`	Prefix
`is_strict`	Flag to indicate when the hop is Strict or Loose
`add_to_top`	Flag to indicate if this hop is to be added to the top of list or not

Output Parameters

None

Return Value

RSVP_ERR_PATH_HOP_EXISTS when the path hop already exists

RSVP_SUCCESS when the hop is successfully added to the ERO path

rsvp_api_path_hop_del

This call deletes a hop from the ERO path.

Syntax

```
int
rsvp_api_path_hop_del (struct rsvp_path *path, struct rsvp_path_ero *ero,
                       struct prefix *p)
```

Input Parameters

`path`	RSVP path
`ero`	ERO path
`p`	Prefix

Output Parameters

None

Return Value

RSVP_SUCCESS when the hop is successfully deleted from the ERO path

rsvp_api_trunk_get_by_name

This call creates a new trunk with the specified name, if not already found.

Syntax

```
struct rsvp_trunk *
rsvp_api_trunk_get_by_name (char *name, u_char family, int *status)
```

Input Parameters

`name`	Trunk name

`family`	Address family; one of AF_INET or AF_INET6

Output Parameters

`status`	Status of the call Success or Error type RSVP_FAILURE when new trunk creation fails or trunk name is duplicate RSVP_ERR_INVALID_NAME when the specified trunk name is invalid RSVP_SUCCESS when call was successful

Return Value

RSVP trunk

rsvp_api_trunk_lookup_by_name

This call looks up RSVP trunk by trunk name.

Syntax

```
struct rsvp_trunk *
rsvp_api_trunk_lookup_by_name (char *name)
```

Input Parameters

`name`	Trunk name

Output Parameters

None

Return Value

RSVP trunk

rsvp_api_delete_trunk

This call deletes the specified trunk.

Syntax

```
void
rsvp_api_delete_trunk (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP Trunk

Output Parameters

`trunk`	RSVP Trunk

Return Value

None

rsvp_api_trunk_reset

This call resets the specified trunk.

Syntax

```
void
rsvp_api_trunk_reset (struct rsvp_trunk *trunk, session_role_t session_role)
```

Input Parameters

`trunk`	RSVP trunk
`session_role`	Session role; one of
	`RSVP_SESSION_ROLE_NONE`
	`RSVP_SESSION_ROLE_INGRESS`
	`RSVP_SESSION_ROLE_TRANSIT`
	`RSVP_SESSION_ROLE_EGRESS`
	`RSVP_SESSION_ROLE_NON_INGRESS`

Output Parameters

None

Return Value

None

rsvp_api_trunk_unset_sessions

This call unsets all configuration.

Syntax

```
void
rsvp_api_trunk_unset_sessions (struct rsvp_trunk *trunk, u_char session_type)
```

Input Parameters

`trunk`	RSVP Trunk
`session_type`	Session type; one of
	`RSVP_SESSION_TYPE_UNDEFINED`
	`RSVP_SESSION_TYPE_PRIMARY`
	`RSVP_SESSION_TYPE_SECONDARY`

Output Parameters

None

Return Value

None

rsvp_api_trunk_ingress_session_get

This call gets an ingress session.

Syntax

```
struct rsvp_session *
rsvp_api_trunk_ingress_session_get (struct rsvp_trunk *trunk,
                                    u_char session_type, bool_t _create)
```

Input Parameters

`trunk`	RSVP Trunk
`session_type`	Session type; one of
	`RSVP_SESSION_TYPE_UNDEFINED`
	`RSVP_SESSION_TYPE_PRIMARY`
	`RSVP_SESSION_TYPE_SECONDARY`
`create`	Flag indicating whether new session is to be created

Output Parameters

None

Return Value

RSVP session

rsvp_api_trunk_default_session_create

This call creates a default session

Syntax

```
struct rsvp_session *
rsvp_api_trunk_default_session_create (struct rsvp_trunk *trunk,
                                       struct prefix *ingress, u_char session_type)
```

Input Parameters

`trunk`	RSVP Trunk
`ingress`	Prefix
`session_type`	Session type; one of
	`RSVP_SESSION_TYPE_UNDEFINED`
	`RSVP_SESSION_TYPE_PRIMARY`
	`RSVP_SESSION_TYPE_SECONDARY`

Output Parameters

None

Return Value

RSVP session

rsvp_api_trunk_map_route

This call maps a route onto this trunk.

Syntax

```
int
rsvp_api_trunk_map_route (struct rsvp_trunk *trunk,
                          struct prefix *p, char *class_str)
```

Input Parameters

`trunk`	RSVP Trunk
`p`	Prefix
`class_str`	Class

Output Parameters

None

Return Value

RSVP_ERR_INVALID_TRUNK when the RSVP trunk is invalid

RSVP_ERR_DSCP_INVALID when the DSCP corresponding to class is invalid

RSVP_ERR_SESSION_MAP_ROUTE when the specified prefix is already configured

RSVP_FAILURE when the call fails

RSVP_SUCCESS when the call is successful

rsvp_api_trunk_unmap_route

This call removes a route from the route map for this trunk.

Syntax

```
int
rsvp_api_trunk_unmap_route (struct rsvp_trunk *trunk,
                            struct prefix *p, char *class_str)
```

Input Parameters

`trunk`	RSVP Trunk
`p`	Prefix
`class_str`	Class

Output Parameters

None

Return Value

RSVP_ERR_INVALID_TRUNK when the RSVP trunk is invalid

RSVP_ERR_DSCP_INVALID when the DSCP corresponding to class is invalid

RSVP_ERR_SESSION_MAP_ROUTE when the given prefix is already configured

RSVP_FAILURE when the call fails

RSVP_SUCCESS when the call is successful

rsvp_api_session_path_set

This call sets the path to be used.

Syntax

```
void
rsvp_api_session_path_set (struct rsvp_session *orig_session,
                           const char *name)
```

Input Parameters

`orig_session`	Session
`name`	Path name

Output Parameters

None

Return Value

None

rsvp_api_session_path_unset

This call unsets the path to use.

Syntax

```
void
rsvp_api_session_path_unset (struct rsvp_session *orig_session, u_char process)
```

Input Parameters

`orig_session`	Session
`process`	Indication whether session is to be updated or not

Output Parameters

None

Return Value

None

rsvp_api_session_set_hop_limit

This call sets the session hop limit for CSPF.

Syntax

```
void
rsvp_api_session_set_hop_limit (struct rsvp_session *orig_session, u_char val)
```

Input Parameters

`orig_session`	Session
`val`	Hop limit

Output Parameters

None

Return Value

None

rsvp_api_session_unset_hop_limit

This call removes the session hop limit for CSPF.

Syntax

```
void
rsvp_api_session_unset_hop_limit (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_hold_priority

This call sets a session hold priority.

Syntax

```
int
rsvp_api_session_set_hold_priority (struct rsvp_session *orig_session, val)
```

Input Parameters

`orig_session`	Session
`val`	Hold priority

Output Parameters

None

Return Value

RSVP_SUCCESS when the call is successful

RSVP_ERR_INVALID_HOLD_PRIORI when there is a problem with the hold priority value being configured

RSVP_FAILURE when any other generic failure occurs

rsvp_api_session_unset_hold_priority

This call removes a session hold priority.

Syntax

```
int
rsvp_api_session_unset_hold_priority (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

RSVP_SUCCESS when the call is successful

RSVP_ERR_INVALID_HOLD_PRIORI if there is a problem with the hold priority value being configured

RSVP_FAILURE if any other generic failure occurs

rsvp_api_session_set_setup_priority

This call sets a setup priority for the session.

Syntax

```
int
rsvp_api_session_set_setup_priority (struct rsvp_session *orig_session, u_char val)
```

Input Parameters

`orig_session`	Session
`val`	Setup priority

Output Parameters

None

Return Value

RSVP_ERR_INVALID_SETUP_PRIORI when the specified setup priority is invalid

RSVP_FAILURE when the call fails

RSVP_SUCCESS when the call is successful

rsvp_api_session_unset_setup_priority

This call removes the session setup priority.

Syntax

```
int
rsvp_api_session_unset_setup_priority (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_retry_count

This call sets a retry count for this session.

Syntax

```
void
rsvp_api_session_set_retry_count (struct rsvp_session *session, u_int16_t val)
```

Input Parameters

`session`	Session
`val`	Retry count

Output Parameters

None

Return Value

None

rsvp_api_session_unset_retry_count

This call removes the retry count for this session.

Syntax

```
void
rsvp_api_session_unset_retry_count (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_retry_interval

This call sets a retry interval for this session.

Syntax

```
void
rsvp_api_session_set_retry_interval (struct rsvp_session *session, u_int16_t val)
```

Input Parameters

`session`	Session
`val`	Retry interval

Output Parameters

None

Return Value

None

rsvp_api_session_unset_retry_interval

This call resets the retry interval for the session.

Syntax

```
void
rsvp_api_session_unset_retry_interval (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_cspf_retry_count

This call sets a retry count for the session for CSPF.

Syntax

```
void
rsvp_api_session_set_cspf_retry_count (struct rsvp_session *session, u_int16_t val)
```

Input Parameters

`session`	Session
`val`	Retry count

Output Parameters

None

Return Value

None

rsvp_api_session_unset_cspf_retry_count

This call removes the retry count for the session for CSPF.

Syntax

```
void
rsvp_api_session_unset_cspf_retry_count (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_cspf_retry_interval

This call sets a retry interval for the session for CSPF.

Syntax

```
void
rsvp_api_session_set_cspf_retry_interval (struct rsvp_session *session,
                                          u_int16_t val)
```

Input Parameters

`session`	Session
`val`	Retry count

Output Parameters

None

Return Value

None

rsvp_api_session_unset_cspf_retry_interval

This call removes the retry interval for the session for CSPF.

Syntax

```
void
rsvp_api_session_unset_cspf_retry_interval (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_bandwidth

This call sets the bandwidth for this session.

Syntax

```
void
rsvp_api_session_set_bandwidth (struct rsvp_session *orig_session,
                                float32_t bandwidth)
```

Input Parameters

`orig_session`	Session
`bandwidth`	Bandwidth

Output Parameters

None

Return Value

None

rsvp_api_session_unset_bandwidth

This call unsets the session bandwidth.

Syntax

```
void
rsvp_api_session_unset_bandwidth (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_filter

This call sets a filter for the session.

Syntax

```
void
rsvp_api_session_set_filter (struct rsvp_session *orig_session, u_char style)
```

Input Parameters

`orig_session`	Session
`style`	RSVP filter style; one of
	`RSVP_STYLE_UNDEFINED`
	`RSVP_STYLE_FIXED`
	`RSVP_STYLE_SHARED_EXPLICIT`

Output Parameters

None

Return Value

None

rsvp_api_session_unset_filter

This call unsets the filter for the session.

Syntax

```
void
rsvp_api_session_unset_filter (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_set_traffic

This call sets traffic for the session.

Syntax

```
void
rsvp_api_session_set_traffic (struct rsvp_session *orig_session, u_char traffic)
```

Input Parameters

`orig_session`	Session
`traffic`	Traffic value

Output Parameters

None

Return Value

None

rsvp_api_session_unset_traffic

This call unsets the traffic for this session.

Syntax

```
void
rsvp_api_session_unset_traffic (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_enable_cspf

This call enables CSPF for the session.

Syntax

```
void
rsvp_api_session_enable_cspf (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_disable_cspf

This call disables CSPF for the session.

Syntax

```
void
rsvp_api_session_disable_cspf (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_enable_route_record_reuse

This call enables Route Record Reuse for the session.

Syntax

```
void
rsvp_api_session_enable_route_record_reuse (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_disable_route_record_reuse

This call disables Route Record Reuse for a session.

Syntax

```
void
rsvp_api_session_disable_route_record_reuse (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_enable_route_record

This call enables Route Record for a session.

Syntax

```
void
rsvp_api_session_enable_route_record (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_disable_route_record

This call disables Route Record for a session.

Syntax

```
void
rsvp_api_session_disable_route_record (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_enable_label_record

This call enables Label Record for a session.

Syntax

```
void
rsvp_api_session_enable_label_record (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_disable_label_record

This call disables Label Record for a session.

Syntax

```
void
rsvp_api_session_disable_label_record (struct rsvp_session *session)
```

Input Parameters

`session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_enable_affinity_packing

This call enables affinity usage for Session attribute object.

Syntax

```
void
rsvp_api_session_enable_affinity_packing (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_disable_affinity_packing

This call disables affinity usage for the session attribute object.

Syntax

```
void
rsvp_api_session_disable_affinity_packing (struct rsvp_session *orig_session)
```

Input Parameters

`orig_session`	Session

Output Parameters

None

Return Value

None

rsvp_api_session_admin_group_set

This call sets an admin group for a session.

Syntax

```
int
rsvp_api_session_admin_group_set (struct rsvp_session *session,
                                  u_char type, char *name)
```

Input Parameters

`session`	Session

`type`	Type of admin group; one of
	`RSVP_SESSION_AFFINITY_EXCLUDE_ANY`
	`RSVP_SESSION_AFFINITY_INCLUDE_ANY`
	`RSVP_SESSION_AFFINITY_INCLUDE_ALL`

`name`	Admin group name

Output Parameters

None

Return Value

RSVP_ERR_INVALID_ARGS when the Arguments are invalid

RSVP_SUCCESS when the call is successful

RSVP_ERR_INTERNAL when there is an internal error

rsvp_api_session_admin_group_unset

This call removes an existing admin group.

Syntax

```
int
rsvp_api_session_admin_group_unset (struct rsvp_session *session,
                                    u_char type, char *name)
```

Input Parameters

`session`	Session

`type`	Type of admin group; one of
	`RSVP_SESSION_AFFINITY_EXCLUDE_ANY`
	`RSVP_SESSION_AFFINITY_INCLUDE_ANY`

`RSVP_SESSION_AFFINITY_INCLUDE_ALL` `name` Admin group name

Output Parameters

None

Return Value

RSVP_ERR_INVALID_ARGS when the Arguments are invalid

RSVP_SUCCESS when the call is successful

RSVP_ERR_INTERNAL when there is an internal error

rsvp_api_session_set_local_protect

This call enables local protection for a session.

Syntax

```
void
rsvp_api_session_set_local_protect (struct rsvp_session *session)
```

Input Parameters

`session` Session

Output Parameters

None

Return Value

None

rsvp_api_session_unset_local_protect

This call disables local protection for a session.

Syntax

```
void
rsvp_api_session_unset_local_protect (struct rsvp_session *session)
```

Input Parameters

`session` Session

Output Parameters

None

Return Value

None

rsvp_api_trunk_set_ext_tunnel_id

This call sets the extended tunnel ID for a trunk.

Syntax

```
int
rsvp_api_trunk_set_ext_tunnel_id (struct rsvp_trunk *trunk, struct prefix *id)
```

Input Parameters

`trunk`	RSVP trunk
`id`	Extended Tunnel ID

Output Parameters

None

Return Value

`RSVP_ERR_INVALID_ARGS` if Arguments are invalid

`RSVP_SUCCESS` when the call is successful

rsvp_api_trunk_unset_ext_tunnel_id

This call unsets the extended tunnel ID for a trunk.

Syntax

```
void
rsvp_api_trunk_unset_ext_tunnel_id (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_set_update_type

This call sets the update type for a trunk.

Syntax

```
void
rsvp_api_trunk_set_update_type (struct rsvp_trunk *trunk, u_char type)
```

Input Parameters

`trunk`	RSVP trunk
`type`	Update Type

Output Parameters

None

Return Value

None

rsvp_api_trunk_unset_update_type

This call unsets the update type for a trunk.

Syntax

```
void
rsvp_api_trunk_unset_update_type (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_set_ingress

This call sets an ingress address for a given trunk.

Syntax

```
int
rsvp_api_trunk_set_ingress (struct rsvp_trunk *trunk, struct prefix *p)
```

Input Parameters

`trunk`	RSVP trunk
`p`	Prefix

Output Parameters

None

Return Value

`RSVP_ERR_INVALID_ARGS` if Arguments are invalid

`RSVP_SUCCESS` when the call is successful

rsvp_api_trunk_unset_ingress

This call unsets an ingress address for a given trunk

Syntax

```
void
rsvp_api_trunk_unset_ingress (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_set_egress

This call sets an egress address for a given trunk

Syntax

```
int
rsvp_api_trunk_set_egress (struct rsvp_trunk *trunk, struct prefix *p)
```

Input Parameters

`trunk`	RSVP trunk
`p`	Prefix

Output Parameters

None

Return Value

`RSVP_ERR_INVALID_ARGS` if Arguments are invalid

`RSVP_SUCCESS` when the call is successful

rsvp_api_trunk_unset_egress

This call unsets an egress address for a given trunk

Syntax

```
void
rsvp_api_trunk_unset_egress (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_restart_all_sessions

This call restarts all specified sessions for a trunk

Syntax

```
void
rsvp_api_trunk_restart_all_sessions (struct rsvp_trunk *trunk,
                                     session_role_t session_role, u_char session_type)
```

Input Parameters

`trunk`	RSVP trunk
`session_role`	Session Role; one of
	`RSVP_SESSION_ROLE_NONE`
	`RSVP_SESSION_ROLE_INGRESS`
	`RSVP_SESSION_ROLE_TRANSIT`
	`RSVP_SESSION_ROLE_EGRESS`
	`RSVP_SESSION_ROLE_NON_INGRESS`
`session_type`	Session Type; one of
	`RSVP_SESSION_TYPE_UNDEFINED`
	`RSVP_SESSION_TYPE_PRIMARY`
	`RSVP_SESSION_TYPE_SECONDARY`

Output Parameters

None

Return Value

None

rsvp_api_trunk_ingress_reset

This call resets the specified ingress trunk.

Syntax

```
void
rsvp_api_trunk_ingress_reset (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_non_ingress_reset

This call resets a specified non-ingress trunk.

Syntax

```
void
rsvp_api_trunk_non_ingress_reset (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_ingress_restart_all_primary_sessions

This call restarts all primary sessions for a specified ingress trunk.

Syntax

```
static void
rsvp_api_trunk_ingress_restart_all_primary_sessions (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_primary_reset

This call resets all primary sessions in a specified trunk.

Syntax

```
void
rsvp_api_trunk_primary_reset (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_ingress_restart_all_secondary_sessions

This call restarts all secondary sessions for a specified ingress trunk.

Syntax

```
static void
rsvp_api_trunk_ingress_restart_all_secondary_sessions (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_trunk_secondary_reset

This call resets all secondary sessions in a specified trunk.

Syntax

```
void
rsvp_api_trunk_secondary_reset (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	RSVP trunk

Output Parameters

None

Return Value

None

rsvp_api_activate_interface

This call activates an RSVP interface.

Syntax

```
void
rsvp_api_activate_interface (struct rsvp_interface *rsvpif, bool_t process_sessions)
```

Input Parameters

`rsvpif` RSVP Interface

`process_sessions`
Indicates whether sessions are to be processed

Output Parameters

None

Return Value

None

rsvp_api_interface_disable

This call disables an RSVP interface.

Syntax

```
void
rsvp_api_interface_disable (struct rsvp_interface *rsvpif, u_char if_down)
```

Input Parameters

`rsvpif` RSVP Interface

`if_down` Indicates that the interface has been administratively shut down

Output Parameters

None

Return Value

None

rsvp_api_if_max_message_id_set

This call sets the Max Message ID value for an interface.

Syntax

```
void
rsvp_api_if_max_message_id_set (struct rsvp_interface *rsvpif, u_int32_t id)
```

Input Parameters

`rsvpif` RSVP Interface

`id` Max message ID

Output Parameters

None

Return Value

None

rsvp_api_if_max_message_id_unset

This call unsets the Max Message ID value for an interface.

Syntax

```
void
rsvp_api_if_max_message_id_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_enable_refresh_reduction

This call enables Refresh Reduction for an interface.

Syntax

```
void
rsvp_api_if_enable_refresh_reduction (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_disable_refresh_reduction

This call disables Refresh Reduction for an interface.

Syntax

```
void
rsvp_api_if_disable_refresh_reduction (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_enable_message_ack

This call enables message ACK for an interface.

Syntax

```
void
rsvp_api_if_enable_message_ack (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_disable_message_ack

This call disables message ACK for an interface.

Syntax

```
void
rsvp_api_if_disable_message_ack (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_ack_wait_timeout_set

This call sets ACK wait timeout for an interface.

Syntax

```
void
rsvp_api_if_ack_wait_timeout_set (struct rsvp_interface *rsvpif, u_int16_t timeout)
```

Input Parameters

`rsvpif`	RSVP Interface

`timeout`	Ack Wait Timeout

Output Parameters

None

Return Value

None

rsvp_api_if_ack_wait_timeout_unset

This call unsets ACK wait timeout for an interface.

Syntax

```
void
rsvp_api_if_ack_wait_timeout_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_bundle_buffer_size_set

This call sets the Bundle buffer size for an interface.

Syntax

```
void
rsvp_api_if_bundle_buffer_size_set (struct rsvp_interface *rsvpif, u_int16_t size)
```

Input Parameters

`rsvpif`	RSVP Interface
`size`	Bundle buffer size

Output Parameters

None

Return Value

None

rsvp_api_if_bundle_buffer_size_unset

This call resets the Bundle buffer size for an interface.

Syntax

```
void
rsvp_api_if_bundle_buffer_size_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_refresh_time_set

This call sets the Refresh Time interval for an interface.

Syntax

```
void
rsvp_api_if_refresh_time_set (struct rsvp_interface *rsvpif, u_int16_t interval)
```

Input Parameters

`rsvpif`	RSVP Interface
`interval`	Refresh time interval

Output Parameters

None

Return Value

None

rsvp_api_if_refresh_time_unset

This call unsets the Refresh Time interval for an interface.

Syntax

```
void
rsvp_api_if_refresh_time_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_keep_multiplier_set

This call sets the interface Keep Multiplier.

Syntax

```
void
rsvp_api_if_keep_multiplier_set (struct rsvp_interface *rsvpif, u_char k)
```

Input Parameters

`rsvpif`	RSVP Interface
`k`	Keep Multiplier

Output Parameters

None

Return Value

None

rsvp_api_if_keep_multiplier_unset

This call unsets the interface Keep Multiplier.

Syntax

```
void
rsvp_api_if_keep_multiplier_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_enable_hello

This call enables sending of Hello messages for an interface.

Syntax

```
void
rsvp_api_if_enable_hello (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_disable_hello

This call disables sending of Hello messages for an interface.

Syntax

```
void
rsvp_api_if_disable_hello (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_hello_interval_set

This call sets the Hello interval for an interface.

Syntax

```
void
rsvp_api_if_hello_interval_set (struct rsvp_interface *rsvpif, u_int16_t val)
```

Input Parameters

`rsvpif`	RSVP Interface
`val`	Hello interval

Output Parameters

None

Return Value

None

rsvp_api_if_hello_interval_unset

This call unsets the Hello interval for an interface.

Syntax

```
void
```

```
rsvp_api_if_hello_interval_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_hello_timeout_set

This call sets the Hello Timeout for an interface.

Syntax

```
void
rsvp_api_if_hello_timeout_set (struct rsvp_interface *rsvpif, u_int16_t val)
```

Input Parameters

`rsvpif`	RSVP Interface
`val`	Hello Timeout

Output Parameters

None

Return Value

None

rsvp_api_if_hello_timeout_unset

This call unsets the Hello Timeout for an interface.

Syntax

```
void
rsvp_api_if_hello_timeout_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_junos_hello_set

This call sets the use of JunOS Hello types for Hello messages exchanged via this interface.

Syntax

```
void
rsvp_api_if_junos_hello_set (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_if_junos_hello_unset

This call stops the use of JunOS Hello types for Hello messages exchanged via this interface.

Syntax

```
void
rsvp_api_if_junos_hello_unset (struct rsvp_interface *rsvpif)
```

Input Parameters

`rsvpif`	RSVP Interface

Output Parameters

None

Return Value

None

rsvp_api_create_rsvp_instance

This call creates a new RSVP instance.

Syntax

```
int
rsvp_api_create_rsvp_instance ()
```

Input Parameters

None

Output Parameters

None

Return Value

RSVP_SUCCESS if successfully created

RSVP_FAILURE if not created

rsvp_api_delete_rsvp_instance

This call deletes the RSVP instance.

Syntax

```
void
rsvp_api_delete_rsvp_instance ()
```

Input Parameters

None

Output Parameters

None

Return Value

None

rsvp_api_disable_cspf

This call disables CSPF usage for RSVP.

Syntax

```
void
rsvp_api_disable_cspf (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_enable_cspf

This call enables CSPF usage for RSVP.

Syntax

```
void
rsvp_api_enable_cspf (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_disable_bundle_send

This call disables sending of Bundle Messages for a system.

Syntax

```
void
rsvp_api_disable_bundle_send (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_enable_bundle_send

This call enables the sending of Bundle Messages for a system.

Syntax

```
void
rsvp_api_enable_bundle_send (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_enable_refresh_reduction

This call enables the Refresh Reduction feature.

Syntax

```
void
rsvp_api_enable_refresh_reduction (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_disable_refresh_reduction

This call disables the Refresh Reduction feature.

Syntax

```
void
rsvp_api_disable_refresh_reduction (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_enable_message_ack

This call enables Message Ack.

Syntax

```
void
rsvp_api_enable_message_ack (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_disable_message_ack

This call disables Message Ack.

Syntax

```
void
rsvp_api_disable_message_ack (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_no_php

This call disables PHP on a router.

Syntax

```
void
rsvp_api_no_php (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_php

This call enables PHP on a router.

Syntax

```
void
rsvp_api_php (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_statistics_reset

This call resets the RSVP statistics.

Syntax

```
void
rsvp_api_statistics_reset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_ack_wait_timeout_set

This call sets an Ack Wait Timeout.

Syntax

```
void
rsvp_api_ack_wait_timeout_set (struct rsvp *rsvp, u_int16_t timeout)
```

Input Parameters

`rsvp`	RSVP instance
`timeout`	Ack Wait Timeout

Output Parameters

None

Return Value

None

rsvp_api_ack_wait_timeout_unset

This call unsets an Ack Wait Timeout.

Syntax

```
void
rsvp_api_ack_wait_timeout_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_refresh_time_set

This call sets the Refresh Time Interval.

Syntax

```
void
rsvp_api_refresh_time_set (struct rsvp *rsvp, u_int16_t interval)
```

Input Parameters

`rsvp`	RSVP instance
`interval`	Refresh time interval

Output Parameters

None

Return Value

None

rsvp_api_refresh_time_unset

This call unsets the Refresh Time Interval.

Syntax

```
void
rsvp_api_refresh_time_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_keep_multiplier_set

This call sets the Keep Multiplier constant.

Syntax

```
void
rsvp_api_keep_multiplier_set (struct rsvp *rsvp, u_char k)
```

Input Parameters

`rsvp`	RSVP instance
`k`	Keep Multiplier

Output Parameters

None

Return Value

None

rsvp_api_keep_multiplier_unset

This call unsets the Keep Multiplier constant.

Syntax

```
void
rsvp_api_keep_multiplier_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_loop_detection_set

This call sets Loop Detection.

Syntax

```
void
rsvp_api_loop_detection_set (struct rsvp *rsvp, u_char val)
```

Input Parameters

`rsvp`	RSVP instance

`val`	Value of loop detection flag; one of
	`RSVP_LOOP_DETECT_ENABLED`
	`RSVP_LOOP_DETECT_DISABLED`
	`RSVP_LOOP_DETECT_ENABLED_ALL`

Output Parameters

None

Return Value

None

rsvp_api_loop_detection_unset

This call unsets Loop Detection.

Syntax

```
void
rsvp_api_loop_detection_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_hello_enable

This call enables sending of Hello messages.

Syntax

```
void
rsvp_api_hello_enable (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_hello_disable

This call disables sending of Hello messages.

Syntax

```
void
rsvp_api_hello_disable (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_hello_interval_set

This call sets the Hello Interval.

Syntax

```
int
rsvp_api_hello_interval_set (struct rsvp *rsvp, u_int16_t val)
```

Input Parameters

`rsvp`	RSVP instance
`val`	Hello interval

Output Parameters

None

Return Value

RSVP_SUCCESS when the call is successful

RSVP_ERR_INVALID_HELLO_INTERVAL when the hello interval is invalid

rsvp_api_hello_interval_unset

This call unsets the Hello Interval

Syntax

```
void
rsvp_api_hello_interval_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_hello_timeout_set

This call sets the Hello Timeout.

Syntax

```
int
rsvp_api_hello_timeout_set (struct rsvp *rsvp, u_int16_t val)
```

Input Parameters

`rsvp`	RSVP instance
`val`	Hello interval

Output Parameters

None

Return Value

RSVP_SUCCESS when the call is successful

RSVP_ERR_INVALID_HELLO_TIMEOUT when the hello timeout is invalid

rsvp_api_hello_timeout_unset

This call unsets the Hello Timeout.

Syntax

```
void
rsvp_api_hello_timeout_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_ingress_set

This call configures Ingress.

Syntax

```
void
rsvp_api_ingress_set (struct rsvp *rsvp, struct prefix *p)
```

Input Parameters

`rsvp`	RSVP instance
`p`	Prefix

Output Parameters

None

Return Value

None

rsvp_api_ingress_unset

This call unsets Ingress.

Syntax

```
void
rsvp_api_ingress_unset (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_neighbor_delete_by_prefix

This call deletes a passed-in neighbor.

Syntax

```
void
rsvp_api_neighbor_delete_by_prefix (struct prefix *p)
```

Input Parameters

`p`	Prefix

Output Parameters

None

Return Value

None

rsvp_api_neighbor_lookup

This call gets an RSVP neighbor.

Syntax

```
struct rsvp_neighbor *
rsvp_api_neighbor_lookup (struct prefix *p)
```

Input Parameters

`p` Prefix

Output Parameters

None

Return Value

RSVP neighbor

rsvp_api_neighbor_create

This call explicitly creates a neighbor.

Syntax

```
int
rsvp_api_neighbor_create (struct prefix *p)
```

Input Parameters

`p` Prefix

Output Parameters

None

Return Value

RSVP_ERR_NEIGHBOR when the address is the same as this router's

RSVP_SUCCESS when the call is successful

RSVP_FAILURE when the call is not successful

rsvp_api_no_refresh_path_parsing

This call unsets refresh path parsing.

Syntax

```
void
rsvp_api_no_refresh_path_parsing ()
```

Input Parameters

None

Output Parameters

None

Return Value

None

rsvp_api_refresh_path_parsing

This call sets Refresh Path parsing.

Syntax

```
void
rsvp_api_refresh_path_parsing (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_no_refresh_resv_parsing

This call unsets refresh RESV parsing.

Syntax

```
void
rsvp_api_no_refresh_resv_parsing (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

rsvp_api_refresh_resv_parsing

This call sets Refresh RESV parsing.

Syntax

```
void
rsvp_api_refresh_resv_parsing (struct rsvp *rsvp)
```

Input Parameters

`rsvp`	RSVP instance

Output Parameters

None

Return Value

None

CHAPTER 14 Management Information Base API

ZebOS-XP supports tables defined in RFC 3812: Multiprotocol Label Switching (MPLS) Traffic Engineering (TE) Management Information Base (MIB).

Supported Tables

mplsTunnelScalarGroup

This table contains scalar objects needed to implement MPLS tunnels.

Object type	Syntax	Access	Functions
mplsTunnelActive	Unsigned32	read-only	rsvp_snmp_get_tn_active
mplsTunnelConfigured	Unsigned32	read-only	rsvp_snmp_get_tn_configured
mplsTunnelMaxHops	Unsigned32	read-only	rsvp_snmp_get_tn_max_hops
mplsTunnelNotificationMaxRate	Unsigned32	read-write	rsvp_snmp_get_tn_notify_max_rate rsvp_snmp_set_tn_notify_max_rate
mplsTunnelTEDistProto	BITS	read-only	rsvp_snmp_get_tn_te_dist_proto

mplsTunnelTable

This table allows new MPLS tunnels to be created between an MPLS LSR and a remote endpoint, and existing tunnels to be reconfigured or removed. Only point-to-point tunnels are supported, although multipoint-to-point and point-to-multipoint connections are supported by an LSR acting as a cross-connect. Each MPLS tunnel can thus have one out-segment originating at an LSR and/or one in-segment terminating at that LSR.

Object type	Syntax	Access	Functions
mplsTunnelName	SnmpAdminString	read-create	rsvp_snmp_get_tn_name rsvp_snmp_get_next_tn_name rsvp_snmp_set_tn_name
mplsTunnelDescr	SnmpAdminString	read-create	rsvp_snmp_get_tn_descr rsvp_snmp_get_next_tn_descr rsvp_snmp_set_tn_descr
mplsTunnelIsIf	TruthValue	read-create	rsvp_snmp_get_tn_is_if rsvp_snmp_get_next_tn_is_if rsvp_snmp_set_tn_is_if
mplsTunnelIfIndex	InterfaceIndexOrZero	read-only	rsvp_snmp_get_tn_if_ix rsvp_snmp_get_next_tn_if_ix

Object type	Syntax	Access	Functions
mplsTunnelOwner	MplsOwner	read-only	rsvp_snmp_get_tn_owner rsvp_snmp_get_next_tn_owner
mplsTunnelRole	INTEGER	read-create	rsvp_snmp_get_tn_role rsvp_snmp_get_next_tn_role rsvp_snmp_set_tn_role
mplsTunnelXCPointer	RowPointer	read-create	rsvp_snmp_get_tn_xc_pointer rsvp_snmp_get_next_tn_xc_pointer rsvp_snmp_set_tn_xc_pointer
mplsTunnelSignallingProto	INTEGER	read-create	rsvp_snmp_get_tn_sig_proto rsvp_snmp_get_next_tn_sig_proto rsvp_snmp_set_tn_sig_proto
mplsTunnelSetupPrio	Integer32	read-create	rsvp_snmp_get_tn_setup_prio rsvp_snmp_get_next_tn_setup_prio rsvp_snmp_set_tn_setup_priority
mplsTunnelHoldingPrio	Integer32	read-create	rsvp_snmp_get_tn_hold_prio rsvp_snmp_get_next_tn_hold_prio rsvp_snmp_set_tn_hold_priority
mplsTunnelSessionAttributes	BITS	read-create	rsvp_snmp_get_tn_sa rsvp_snmp_get_next_tn_sa rsvp_snmp_set_tn_sa
mplsTunnelLocalProtectInUse	TruthValue	read-create	rsvp_snmp_get_tn_lopro_inuse rsvp_snmp_get_next_tn_lopro_inuse rsvp_snmp_set_local_protect
mplsTunnelResourcePointer	RowPointer	read-create	rsvp_snmp_get_tn_rs_pointer rsvp_snmp_get_next_tn_rs_pointer rsvp_snmp_set_tn_rs_pointer
mplsTunnelPrimaryInstance	MplsTunnelInstanceIndex	read-only	rsvp_snmp_get_tn_prim_inst rsvp_snmp_get_next_tn_prim_inst
mplsTunnelInstancePriority	Unsigned32	read-create	rsvp_snmp_get_tn_inst_prio rsvp_snmp_get_next_tn_inst_prio rsvp_snmp_set_tn_inst_prio
mplsTunnelHopTableIndex	MplsPathIndexOrZero	read-create	rsvp_snmp_get_tn_hop_table_ix rsvp_snmp_get_next_tn_hop_table_ix rsvp_snmp_set_tn_hop_table_ix
mplsTunnelPathInUse	MplsPathIndexOrZero	read-create	rsvp_snmp_get_tn_path_inuse rsvp_snmp_get_next_tn_path_inuse rsvp_snmp_set_tn_path_inuse
mplsTunnelARHopTableIndex	MplsPathIndexOrZero	read-only	rsvp_snmp_get_tn_arhop_table_ix rsvp_snmp_get_next_tn_arhop_table_ix
mplsTunnelCHopTableIndex	MplsPathIndexOrZero	read-only	rsvp_snmp_get_tn_chop_table_ix rsvp_snmp_get_next_tn_chop_table_ix

Object type	Syntax	Access	Functions
mplsTunnelIncludeAnyAffinity	MplsTunnelAffinity	read-create	rsvp_snmp_get_tn_incl_any rsvp_snmp_get_next_tn_incl_any rsvp_snmp_set_tn_incl_any
mplsTunnelIncludeAllAffinity	MplsTunnelAffinity	read-create	rsvp_snmp_get_tn_incl_all rsvp_snmp_get_next_tn_incl_all rsvp_snmp_set_tn_incl_all
mplsTunnelExcludeAnyAffinity	MplsTunnelAffinity	read-create	rsvp_snmp_get_tn_excl_any rsvp_snmp_get_next_tn_excl_any rsvp_snmp_set_tn_excl_any
mplsTunnelTotalUpTime	TimeTicks	read-only	rsvp_snmp_get_tn_total_uptime rsvp_snmp_get_next_tn_total_uptime
mplsTunnelInstanceUpTime	TimeTicks	read-only	rsvp_snmp_get_tn_inst_uptime rsvp_snmp_get_next_tn_inst_uptime
mplsTunnelPrimaryUpTime	TimeTicks	read-only	rsvp_snmp_get_tn_prim_timeup rsvp_snmp_get_next_tn_prim_timeup
mplsTunnelPathChanges	Counter32	read-only	rsvp_snmp_get_tn_path_chan rsvp_snmp_get_next_tn_path_chan
mplsTunnelLastPathChange	TimeTicks	read-only	rsvp_snmp_get_tn_last_path_chan rsvp_snmp_get_next_tn_last_path_chan
mplsTunnelCreationTime	TimeStamp	read-only	rsvp_snmp_get_tn_creation_time rsvp_snmp_get_next_tn_creation_time
mplsTunnelStateTransitions	Counter32	read-only	rsvp_snmp_get_tn_state_trans rsvp_snmp_get_next_tn_state_trans
mplsTunnelAdminStatus	INTEGER	read-only	rsvp_snmp_get_tn_admin_status rsvp_snmp_get_next_tn_admin_status rsvp_snmp_set_tn_admin_status
mplsTunnelOperStatus	INTEGER	read-only	rsvp_snmp_get_tn_oper_status rsvp_snmp_get_next_tn_oper_status
mplsTunnelRowStatus	RowStatus	read-create	rsvp_snmp_get_tn_row_status rsvp_snmp_get_next_tn_row_status rsvp_snmp_set_tn_row_status
mplsTunnelStorageType	StorageType	read-create	rsvp_snmp_get_tn_st_type rsvp_snmp_get_next_tn_st_type rsvp_snmp_set_tn_st_type

mplsTunnelResourceTable

This table indicates the resources required for a tunnel. Multiple tunnels can share the same resources by pointing to the same entry in this table. Tunnels that do not share resources must point to separate entries in this table.

Object type	Syntax	Access	Functions
mplsTunnelResourceIndexNext	Unsigned32	read-only	rsvp_snmp_get_tn_resource_ix_next
mplsTunnelResourceMaxRate	MplsBitRate	read-create	rsvp_snmp_get_tn_resource_maxrate rsvp_snmp_get_next_tn_resource_maxrate rsvp_snmp_set_tn_resource_maxrate
mplsTunnelResourceMeanRate	MplsBitRate	read-create	rsvp_snmp_get_tn_resource_meanrate rsvp_snmp_get_next_tn_resource_meanrate rsvp_snmp_set_tn_resource_meanrate
mplsTunnelResourceMaxBurstSize	MplsBurstSize	read-create	rsvp_snmp_get_tn_resource_maxburstsize rsvp_snmp_get_next_tn_resource_maxburstsize rsvp_snmp_set_tn_resource_maxburstsize
mplsTunnelResourceMeanBurstSize	MplsBurstSize	read-create	rsvp_snmp_get_tn_resource_meanburstsize rsvp_snmp_get_next_tn_resource_meanburstsize rsvp_snmp_set_tn_resource_meanburstsize
mplsTunnelResourceExBurstSize	MplsBurstSize	read-create	rsvp_snmp_get_tn_resource_exburstsize rsvp_snmp_get_next_tn_resource_exburstsize rsvp_snmp_set_tn_resource_exburstsize
mplsTunnelResourceFrequency	INTEGER	read-create	rsvp_snmp_get_tn_resource_frequency rsvp_snmp_get_next_tn_resource_frequency rsvp_snmp_set_tn_resource_frequency
mplsTunnelResourceWeight	Unsigned32	read-create	rsvp_snmp_get_tn_resource_weight rsvp_snmp_get_next_tn_resource_weight rsvp_snmp_set_tn_resource_weight
mplsTunnelResourceRowStatus	RowStatus	read-create	rsvp_snmp_get_tn_resource_rowstatus rsvp_snmp_get_next_tn_resource_rowstatus rsvp_snmp_set_tn_resource_rowstatus
mplsTunnelResourceStorageType	StorageType	read-create	rsvp_snmp_get_tn_resource_storagetype rsvp_snmp_get_next_tn_resource_storagetype rsvp_snmp_set_tn_resource_storagetype

mplsTunnelHopTable

This table indicates the hops, strict or loose, for an MPLS tunnel defined in mplsTunnelTable, when it is established via signalling. Multiple tunnels can share the same hops by pointing to the same entry in this table.

At transit LSRs, this table contains the hops, strict or loose, that apply to the downstream part of this tunnel only. This corresponds to the requested path received through the signaling protocol.

Object type	Syntax	Access	Functions
mplsTunnelHopListIndexNext	MplsPathIndexOrZero	read-only	rsvp_snmp_get_tn_hop_ix_next
mplsTunnelHopAddrType	TeHopAddressType	read-create	rsvp_snmp_get_hop_addr_type rsvp_snmp_get_next_hop_addr_type rsvp_snmp_set_tn_hop_addr_type
mplsTunnelHopIpAddr	TeHopAddress	read-create	rsvp_snmp_get_hop_ip_addr rsvp_snmp_get_next_hop_ip_addr rsvp_snmp_set_tn_hop_ip_addr
mplsTunnelHopIpPrefixLen	InetAddressPrefixLength	read-create	rsvp_snmp_get_hop_ip_pref_len rsvp_snmp_get_next_hop_ip_pref_len rsvp_snmp_set_tn_hop_ip_pref_len
mplsTunnelHopAsNumber	TeHopAddressAS	read-create	rsvp_snmp_get_hop_as_num rsvp_snmp_get_next_hop_as_num rsvp_snmp_set_tn_hop_as_num
mplsTunnelHopAddrUnnum	TeHopAddressUnnum	read-create	rsvp_snmp_get_hop_addr_un_num rsvp_snmp_get_next_hop_addr_un_num rsvp_snmp_set_hop_addr_un_num
mplsTunnelHopLspId	MplsLSPID	read-create	rsvp_snmp_get_hop_lspid rsvp_snmp_get_next_hop_lspid rsvp_snmp_set_tn_hop_lspid
mplsTunnelHopType	INTEGER	read-create	rsvp_snmp_get_hop_type rsvp_snmp_get_next_hop_type rsvp_snmp_set_tn_hop_type
mplsTunnelHopInclude	TruthValue	read-create	rsvp_snmp_get_hop_incl_excl rsvp_snmp_get_next_hop_incl_excl rsvp_snmp_set_tn_hop_incl_excl
mplsTunnelHopPathOptionName	SnmpAdminString	read-create	rsvp_snmp_get_hop_path_op_name rsvp_snmp_get_next_hop_path_op_name rsvp_snmp_set_tn_hop_path_set
mplsTunnelHopEntryPathComp	INTEGER	read-create	rsvp_snmp_get_hop_entry_path_comp rsvp_snmp_get_next_hop_entry_path_comp rsvp_snmp_set_tn_hop_entry_path_comp
mplsTunnelHopRowStatus	RowStatus	read-create	rsvp_snmp_get_hop_row_status rsvp_snmp_get_next_hop_row_status rsvp_snmp_set_tn_hop_row_status
mplsTunnelHopStorageType	StorageType	read-create	rsvp_snmp_get_hop_st_type rsvp_snmp_get_next_hop_st_type rsvp_snmp_set_tn_hop_st_type

mplsTunnelPerfTable

This table provides several counters to measure the performance of the MPLS tunnels. This table augments mplsTunnelTable.

Object type	Syntax	Access	Functions
mplsTunnelPerfPackets	Counter32	read-only	rsvp_get_tn_perf_pkts rsvp_snmp_get_next_tn_perf_pkts rsvp_snmp_get_tn_perf_pkts
mplsTunnelPerfHCPackets	Counter64	read-only	rsvp_get_tn_perf_hc_pkts rsvp_snmp_get_next_tn_perf_hc_pkts rsvp_snmp_get_tn_perf_hc_pkts
mplsTunnelPerfErrors	Counter32	read-only	rsvp_get_tn_perf_errors rsvp_snmp_get_next_tn_perf_errors rsvp_snmp_get_tn_perf_errors
mplsTunnelPerfBytes	Counter32	read-only	rsvp_get_tn_perf_bytes rsvp_snmp_get_next_tn_perf_bytes rsvp_snmp_get_tn_perf_bytes
mplsTunnelPerfHCBytes	Counter64	read-only	rsvp_get_tn_perf_hc_bytes rsvp_snmp_get_next_tn_perf_hc_bytes rsvp_snmp_get_tn_perf_hc_bytes

Notifications

ZebOS-XP generates notifications for mplsTunnelUp, mplsTunnelDown, and mplsTunnelRerouted notifications.

Object type	Syntax	Access	Functions
mplsTunnelNotificationEnable	TruthValue	read-write	rsvp_snmp_get_tn_notify_en rsvp_snmp_set_tn_notify_en
mplsTunnelUp	n/a	n/a	rsvp_tn_up
mplsTunnelDown	n/a	n/a	rsvp_tn_down
mplsTunnelRerouted	n/a	n/a	rsvp_tn_reroute

API

rsvp_get_tn_perf_bytes

This function gets the number of tunnel bytes transmitted.

Syntax

```
u_int32_t
rsvp_get_tn_perf_bytes (struct rsvp_session *session)
```

Input Parameters

`session` RSVP session

Output Parameters

None

Return Value

Number of bytes transmitted

rsvp_get_tn_perf_errors

This function gets the number of error tunnel packets.

Syntax

```
u_int32_t
rsvp_get_tn_perf_errors (struct rsvp_session *session)
```

Input Parameters

`session` RSVP session

Output Parameters

None

Return Value

Number of error packets

rsvp_get_tn_perf_hc_bytes

This function gets the number of high capacity (64-bit) tunnel bytes transmitted.

Note: The default implementation of this function always returns zero. You need to implement his function according to your platform requirements.

Syntax

```
u_int32_t
rsvp_get_tn_perf_hc_bytes (struct rsvp_session *session)
```

Input Parameters

`session` RSVP session

Output Parameters

None

Return Value

Always zero

rsvp_get_tn_perf_hc_pkts

This function gets the number of high capacity (64-bit) tunnel packets transmitted.

Syntax

```
u_int32_t
rsvp_get_tn_perf_hc_pkts (struct rsvp_session *session)
```

Input Parameters

`session`	RSVP session

Output Parameters

None

Return Value

Number of high capacity packets transmitted

rsvp_get_tn_perf_pkts

This function gets the number of tunnel packets transmitted.

Syntax

```
u_int32_t
rsvp_get_tn_perf_pkts (struct rsvp_session *session)
```

Input Parameters

`session`	RSVP session

Output Parameters

None

Return Value

Number of packets transmitted

rsvp_snmp_get_hop_addr_type

This function gets the address type of this tunnel hop.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_addr_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                             u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop

Output Parameters

ret	This constant from rsvpd/rsvp_api.h:
RSVP_IPV4	

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_addr_un_num

This function gets the unnumbered hop address.

Note: There is no support for mplsTunnelHopAddrType unnum (4) in the backend, so ret is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_addr_un_num (u_int32_t *ero_ix, u_int32_t *path_ix,
                               u_int32_t *hop_ix, int *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	There is no support for mplsTunnelHopAddrType unnum (4) in the backend, so this parameter is always zero

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_as_num

This function gets the autonomous system number of this hop.

Note: There is no support for mplsTunnelHopAddrType asnumber (3) in the backend, so ret is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_as_num (u_int32_t *ero_ix, u_int32_t *path_ix,
                          u_int32_t *hop_ix,
                          u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path

hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	There is no support for mplsTunnelHopAddrType asnumber (3) in the backend, so this parameter is always zero

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_entry_path_comp

This function gets whether CSPF will compute the remainder of this path.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_entry_path_comp (u_int32_t *ero_ix, u_int32_t *path_ix,
                                   u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	One of these constants from rsvpd/rsvp_snmp.h: RSVP_SNMP_PATH_COMPUTED RSVP_SNMP_PATH_NOT_COMPUTED

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_incl_excl

This function gets whether this hop is included or excluded in the tunnel's path.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_incl_excl (u_int32_t *ero_ix, u_int32_t *path_ix,
                             u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	This constant from rsvpd/rsvp_snmp.h:
	RSVP_SNMP_HOP_INCLUDE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_ip_addr

This function gets the IP address of this tunnel hop.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_ip_addr (u_int32_t *ero_ix, u_int32_t *path_ix,
                           u_int32_t *hop_ix, char **ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	IP address

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_ip_pref_len

The call gets the prefix length for this hop.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_ip_pref_len (u_int32_t *ero_ix, u_int32_t *path_ix,
                               u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	Prefix length

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel hop is not found

rsvp_snmp_get_hop_lspid

This function gets the LSP identifier of this hop.

Note: There is no support for `mplsTunnelHopAddrType lspid` (5) in the backend, so `ret` is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_lspid (u_int32_t *ero_ix, u_int32_t *path_ix,
                         u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop

Output Parameters

`ret`	There is no support for `mplsTunnelHopAddrType lspid` (5) in the backend, so this parameter is always zero

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel hop is not found

rsvp_snmp_get_hop_path_op_name

This function gets the path option name which is a description of this series of hops.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_path_op_name (u_int32_t *ero_ix, u_int32_t *path_ix,
                                u_int32_t *hop_ix, char **ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop

Output Parameters

`ret`	Path option name

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_row_status

This function gets the row status of this hop.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_row_status (u_int32_t *ero_ix, u_int32_t *path_ix,
                              u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	One of these constants from rsvpd/rsvp_snmp.h:

RSVP_SNMP_ROW_STATUS_ACTIVE

RSVP_SNMP_ROW_STATUS_NOTINSERVICE

RSVP_SNMP_ROW_STATUS_NOTREADY

RSVP_SNMP_ROW_STATUS_CREATEANDGO

RSVP_SNMP_ROW_STATUS_CREATEANDWAIT

RSVP_SNMP_ROW_STATUS_DESTROY

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_st_type

This function gets the storage type of this hop.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_st_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                           u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	This constant from rsvpd/rsvp_snmp.h:

RSVP_SNMP_ST_TYPE_VOLATILE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_hop_type

This function gets whether this tunnel hop is routed in a strict or loose fashion.

Syntax

```
u_int32_t
rsvp_snmp_get_hop_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                        u_int32_t *hop_ix, u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop

Output Parameters

ret	One of these constants from rsvpd/rsvp_api.h:

RSVP_HOP_TYPE_IS_STRICT

RSVP_HOP_TYPE_IS_LOOSE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_addr_type

This function gets the address type of the next tunnel hop.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_addr_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                                  u_int32_t *hop_ix,
                                  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
ix_len	Index length

Output Parameters

ero_ix	Primary index of the next explicit route object
path_ix	Secondary index identifying the next group of hops for a configured path
hop_ix	Secondary index identifying the next hop
ret	This constant from rsvpd/rsvp_snmp.h: RSVP_SNMP_ADDR_TYPE_IPADDR

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_addr_un_num

This function gets the Unnumbered Hop Address of the next hop.

Note: There is no support for mplsTunnelHopAddrType unnum (4) in the backend, so ret is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_addr_un_num (u_int32_t *ero_ix,
                                    u_int32_t *path_ix, u_int32_t *hop_ix,
                                    u_int32_t ix_len, int *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
ix_len	Index length

Output Parameters

ero_ix	Primary index of the next explicit route object
path_ix	Secondary index identifying the next group of hops for a configured path
hop_ix	Secondary index identifying the next hop
ret	There is no support for mplsTunnelHopAddrType unnum (4) in the backend, so this parameter is always zero

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_as_num

This function gets the autonomous system number of the next hop.

Note: There is no support for mplsTunnelHopAddrType asnumber (3) in the backend, so ret is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_as_num (u_int32_t *ero_ix, u_int32_t *path_ix,
                               u_int32_t *hop_ix,
                               u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`ix_len`	Index length

Output Parameters

`ero_ix`	Primary index of the next explicit route object
`path_ix`	Secondary index identifying the next group of hops for a configured path
`hop_ix`	Secondary index identifying the next hop
`ret`	There is no support for `mplsTunnelHopAddrType asnumber` (3) in the backend, so this parameter is always zero

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel hop is not found

rsvp_snmp_get_next_hop_entry_path_comp

This function gets whether CSPF will compute the remainder of the path for the next hop.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_entry_path_comp (u_int32_t *ero_ix, u_int32_t *path_ix,
                                        u_int32_t *hop_ix,
                                        u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`ix_len`	Index length

Output Parameters

`ero_ix`	Primary index of the next explicit route object
`path_ix`	Secondary index identifying the next group of hops for a configured path
`hop_ix`	Secondary index identifying the next hop
`ret`	One of these constants from `rsvpd/rsvp_snmp.h`:

RSVP_SNMP_PATH_COMPUTED

RSVP_SNMP_PATH_NOT_COMPUTED

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_incl_excl

This function gets whether next hop is included or excluded in the tunnel's path.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_incl_excl (u_int32_t *ero_ix, u_int32_t *path_ix,
                                  u_int32_t *hop_ix,
                                  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
ix_len	Index length

Output Parameters

ero_ix	Primary index of the next explicit route object
path_ix	Secondary index identifying the next group of hops for a configured path
hop_ix	Secondary index identifying the next hop
ret	This constant from rsvpd/rsvp_snmp.h:

RSVP_SNMP_HOP_INCLUDE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_ip_addr

This function gets the IP address of the next tunnel hop.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_ip_addr (u_int32_t *ero_ix, u_int32_t *path_ix,
                                u_int32_t *hop_ix,
                                u_int32_t ix_len,char **ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
ix_len	Index length

Output Parameters

ero_ix	Primary index of the next explicit route object
path_ix	Secondary index identifying the next group of hops for a configured path
hop_ix	Secondary index identifying the next hop
ret	IP address

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_ip_pref_len

The call gets the appropriate prefix length for next hop.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_ip_pref_len (u_int32_t *ero_ix, u_int32_t *path_ix,
                                    u_int32_t *hop_ix,
                                    u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
ix_len	Index length

Output Parameters

ero_ix	Primary index of the next explicit route object
path_ix	Secondary index identifying the next group of hops for a configured path
hop_ix	Secondary index identifying the next hop
ret	Prefix length

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_lspid

This function gets the LSP identifier of next hop.

Note: There is no support for `mplsTunnelHopAddrType lspid` (5) in the backend, so `ret` is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_lspid (u_int32_t *ero_ix, u_int32_t *path_ix,
                              u_int32_t *hop_ix,  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`ix_len`	Index length

Output Parameters

`ero_ix`	Primary index of the next explicit route object
`path_ix`	Secondary index identifying the next group of hops for a configured path
`hop_ix`	Secondary index identifying the next hop
`ret`	There is no support for `mplsTunnelHopAddrType lspid` (5) in the backend, so this parameter is always zero

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel hop is not found

rsvp_snmp_get_next_hop_path_op_name

This function gets the next path option name which is a description of a series of hops.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_path_op_name (u_int32_t *ero_ix, u_int32_t *path_ix,
                                     u_int32_t *hop_ix, u_int32_t ix_len,char **ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`ix_len`	Index length

Output Parameters

`ero_ix`	Primary index of the next explicit route object
`path_ix`	Secondary index identifying the next group of hops for a configured path

hop_ix	Secondary index identifying the next hop
ret	Path option name

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_row_status

This function gets the row status of next hop.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_row_status (u_int32_t *ero_ix, u_int32_t *path_ix,
                                   u_int32_t *hop_ix,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
ix_len	Index length

Output Parameters

ero_ix	Primary index of the next explicit route object
path_ix	Secondary index identifying the next group of hops for a configured path
hop_ix	Secondary index identifying the next hop
ret	One of these constants from rsvpd/rsvp_snmp.h:

RSVP_SNMP_ROW_STATUS_ACTIVE

RSVP_SNMP_ROW_STATUS_NOTINSERVICE

RSVP_SNMP_ROW_STATUS_NOTREADY

RSVP_SNMP_ROW_STATUS_CREATEANDGO

RSVP_SNMP_ROW_STATUS_CREATEANDWAIT

RSVP_SNMP_ROW_STATUS_DESTROY

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_hop_st_type

This function gets the storage type of the next hop.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_st_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                                u_int32_t *hop_ix, u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`ix_len`	Index length

Output Parameters

`ero_ix`	Primary index of the next explicit route object
`path_ix`	Secondary index identifying the next group of hops for a configured path
`hop_ix`	Secondary index identifying the next hop
`ret`	This constant from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ST_TYPE_VOLATILE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel hop is not found

rsvp_snmp_get_next_hop_type

This function gets the hop type which denotes whether next tunnel hop is routed in a strict or loose fashion.

Syntax

```
u_int32_t
rsvp_snmp_get_next_hop_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                             u_int32_t *hop_ix,
                                       u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`ix_len`	Index length

Output Parameters

`ero_ix`	Primary index of the next explicit route object
`path_ix`	Secondary index identifying the next group of hops for a configured path
`hop_ix`	Secondary index identifying the next hop
`ret`	One of these constant from `rsvpd/rsvp_api.h`:

`RSVP_HOP_TYPE_IS_STRICT`

RSVP_HOP_TYPE_IS_LOOSE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel hop is not found

rsvp_snmp_get_next_tn_admin_status

This function gets the administration status of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_admin_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                    struct pal_in4_addr *ing_id,
                                    struct pal_in4_addr *eg_id,
                                    u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	One of these constants from rsvpd/rsvp_snmp.h or rsvpd/rsvp_api.h:

RSVP_SNMP_TUNNEL_UP

RSVP_SNMP_TUNNEL_DOWN

RSVP_TN_ADMIN_DOWN

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_arhop_table_ix

This function gets the index into the mplsTunnelARHopTable entry that specifies the actual hops traversed by the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_arhop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                      struct pal_in4_addr *ing_id,
                                      struct pal_in4_addr *eg_id,
                                      u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	This parameter is always zero because mplsTunnelARHopTable is not supported

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_chop_table_ix

This function gets the index into the mplsTunnelCHopTable entry that specifies the computed hops traversed by the next tunnel.

Note: This object is not supported, so the `ret` parameter is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_chop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                     struct pal_in4_addr *ing_id,
                                     struct pal_in4_addr *eg_id,
                                     u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	This parameter is always zero because mplsTunnelCHopTable is not supported

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_creation_time

The call gets the SysUpTime when the first instance of the next tunnel came into existence.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_creation_time (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                     struct pal_in4_addr *ing_id,
                                     struct pal_in4_addr *eg_id,
                                     u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	When the first instance of this tunnel came into existence

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_descr

This function gets the description of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_descr (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id,
                             u_int32_t ix_len,char **ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Description of the next tunnel.

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_excl_any

This function gets the exclude-any constraint of the tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_excl_any (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id,
                                u_int32_t ix_len,u_int32_t  *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance

inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	One of these constants from rsvpd/rsvpd.h:
RSVP_TRUE	
RSVP_FALSE	

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_hold_prio

This function gets the holding priority for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_hold_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Holding priority

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_hop_table_ix

This function gets the index into the mplsTunnelHopTable entry that specifies the explicit route hops for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_hop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                    struct pal_in4_addr *ing_id,
                                    struct pal_in4_addr *eg_id,
                                    u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Index into mplsTunnelHopTable

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_if_ix

If `mplsTunnelIsIf` is set to true, then this function gets the next tunnel's LSR-assigned `ifIndex` which corresponds to an entry in this interfaces table.

Note: This object is not supported, so `ret` is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_if_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id,
                             u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	There is no support for this object, so this parameter is always zero

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_incl_all

This function gets the next tunnel's include-all constraint.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_incl_all (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id,
                                u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	One of these constants from rsvpd/rsvpd.h: RSVP_TRUE RSVP_FALSE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_incl_any

This function gets the next tunnel's include-any constraint.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_incl_any (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id,
                                u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	One of these constants from `rsvpd/rsvpd.h`:
	`RSVP_TRUE`
	`RSVP_FALSE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_inst_prio

This function gets the priority of the next tunnel instance within a group of tunnel instances.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_inst_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index

ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Priority; one of these constants from rsvpd/rsvp_snmp.h:

RSVP_INSTANCE_PRIO_PRIMARY
RSVP_INSTANCE_PRIO_SECONDARY

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_inst_uptime

This function gets the total time that the operational status of the next tunnel instance has been up.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_inst_uptime (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                   struct pal_in4_addr *ing_id,
                                   struct pal_in4_addr *eg_id,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Up time

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_is_if

This function gets whether the next tunnel corresponds to an interface represented in the interfaces group table.

Note: This object is not supported, so ret is always RSVP_FALSE.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_is_if (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id, struct pal_in4_addr *eg_id,
                             u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	This constant from rsvpd/rsvpd.h:

RSVP_FALSE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_last_path_chan

This function gets the time since the last path changed for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_last_path_chan (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                      struct pal_in4_addr *ing_id,
                                      struct pal_in4_addr *eg_id,
                                      u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index

inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Time since the last path changed

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_lopro_inuse

This function gets wether the local repair mechanism is in use for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_lopro_inuse (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                   struct pal_in4_addr *ing_id,
                                   struct pal_in4_addr *eg_id,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	One of these constants from rsvpd/rsvp_snmp.h:

RSVP_SNMP_FALSE

RSVP_SNMP_TRUE

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_name

This function gets the canonical name of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_name (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id, struct pal_in4_addr *eg_id,
                            u_int32_t ix_len,char **ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Canonical name of the next tunnel

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_oper_status

This function gets the operating status of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_oper_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                   struct pal_in4_addr *ing_id,
                                   struct pal_in4_addr *eg_id,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index

`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_TUNNEL_UP`

`RSVP_SNMP_TUNNEL_DOWN`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_owner

This function gets the entity that creates and manages the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_owner (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id, struct pal_in4_addr *eg_id,
                             u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	This constant from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_RSVP_OWNER`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_path_chan

This function gets the number of times the path has changed for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_path_chan (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Number of times the path has changed

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_path_inuse

This function gets the index of the path chosen for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_path_inuse (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                  struct pal_in4_addr *ing_id,
                                  struct pal_in4_addr *eg_id,
                                  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Index of the path

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_perf_bytes

This function gets the number of bytes forwarded by the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_perf_bytes (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                  struct pal_in4_addr *ing_id,
                                  struct pal_in4_addr *eg_id,
                                  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Number of bytes forwarded

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_perf_errors

This function gets the number of packets with errors received by the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_perf_errors (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                   struct pal_in4_addr *ing_id,
                                   struct pal_in4_addr *eg_id,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Number of error packets received

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_perf_hc_bytes

This function gets the high capacity counter of number of bytes forwarded by the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_perf_hc_bytes (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                     struct pal_in4_addr *ing_id,
                                     struct pal_in4_addr *eg_id,
                                     u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Number of bytes forwarded

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_perf_hc_pkts

This function gets the high capacity counter for number of packets forwarded by the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_perf_hc_pkts (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                    struct pal_in4_addr *ing_id,
                                       struct pal_in4_addr *eg_id,
                                       u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	The high capacity counter for number of packets forwarded

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_perf_pkts

This function gets number of packets forwarded by the next tunnel instance.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_perf_pkts (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Number of packets forwarded.

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_prim_inst

This function gets the instance index of the primary instance of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_prim_inst (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Primary instance index of the next tunnel

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_prim_timeup

This function gets the total time the primary instance of the next tunnel has been active.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_prim_timeup (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                   struct pal_in4_addr *ing_id,
                                   struct pal_in4_addr *eg_id,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Active time

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_resource_exburstsize

This function gets the excess burst size in bytes of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_exburstsize (int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	Next excess burst size in bytes

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_resource_frequency

This function gets the frequency of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_frequency (int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	Next frequency

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_resource_maxburstsize

This function gets the maximum burst size in bytes of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_maxburstsize (int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	Next maximum burst size in bytes

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_resource_maxrate

This function gets the maximum rate of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_maxrate(int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	Next mean rate

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_next_tn_resource_meanburstsize

This function gets the mean burst size in bytes of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_meanburstsize (int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	Next mean burst size in bytes

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_next_tn_resource_meanrate

This function gets the mean rate for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_meanrate(int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	Next mean rate

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_next_tn_resource_rowstatus

This function gets the row status of the next Tunnel Resource Instance.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_rowstatus (int *rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`rs_ix`	Next resource table index
`ret`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ROW_STATUS_ACTIVE`

`RSVP_SNMP_ROW_STATUS_NOTINSERVICE`

`RSVP_SNMP_ROW_STATUS_NOTREADY`

`RSVP_SNMP_ROW_STATUS_CREATEANDGO`

RSVP_SNMP_ROW_STATUS_CREATEANDWAIT

RSVP_SNMP_ROW_STATUS_DESTROY

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the resource table is not found

rsvp_snmp_get_next_tn_resource_storagetype

This function gets the storage type of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_storagetype (int *rs_ix, u_int32_t *ret)
```

Input Parameters

rs_ix	Resource table index

Output Parameters

rs_ix	Next resource table index
ret	This constant from rsvpd/rsvp_snmp.h:

RSVP_SNMP_ST_TYPE_VOLATILE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the resource table is not found

rsvp_snmp_get_next_tn_resource_weight

This function gets the weight of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_resource_weight (int *rs_ix, u_int32_t *ret)
```

Input Parameters

rs_ix	Resource table index

Output Parameters

rs_ix	Next resource table index
ret	Next weight

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the resource table is not found

rsvp_snmp_get_next_tn_role

This function gets the role that next tunnel entry/instance represents.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_role (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id,
                            u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	One of these constants from the `session_role_type` enum in `rsvpd/rsvpd.h`:

`RSVP_SESSION_ROLE_NONE`

`RSVP_SESSION_ROLE_INGRESS`

`RSVP_SESSION_ROLE_TRANSIT`

`RSVP_SESSION_ROLE_EGRESS`

`RSVP_SESSION_ROLE_NON_INGRESS`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_row_status

This function gets the row status of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_row_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                  struct pal_in4_addr *ing_id,
                                  struct pal_in4_addr *eg_id,
                                  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ROW_STATUS_ACTIVE`

`RSVP_SNMP_ROW_STATUS_NOTINSERVICE`

`RSVP_SNMP_ROW_STATUS_NOTREADY`

`RSVP_SNMP_ROW_STATUS_CREATEANDGO`

`RSVP_SNMP_ROW_STATUS_CREATEANDWAIT`

`RSVP_SNMP_ROW_STATUS_DESTROY`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_rs_pointer

This function gets the pointer to the resource table of next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_rs_pointer (u_int32_t *tn_ix,
                                  u_int32_t *inst_ix,
                                  struct pal_in4_addr *ing_id,
                                  struct pal_in4_addr *eg_id,
                                  u_int32_t ix_len,char **ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Pointer to the resource table

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_sa

This function gets the session attributes of the next tunnel.

Note: This object is not supported, so the ret parameter is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_sa (u_int32_t *tn_ix, u_int32_t *inst_ix,
                          struct pal_in4_addr *ing_id,
                          struct pal_in4_addr *eg_id,
                          u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	This object is not supported, so this parameter is always zero

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_setup_prio

This function gets the setup priority of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_setup_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                  struct pal_in4_addr *ing_id,
                                  struct pal_in4_addr *eg_id,
                                  u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	Setup priority

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found.

rsvp_snmp_get_next_tn_sig_proto

This function gets the signaling protocol that was used to set up the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_sig_proto (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	This constant defined in rsvpd/rsvp_snmp.h: RSVP_SNMP_RSVP_PROTO

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_state_trans

This function gets the number of times the state of next tunnel instance has changed.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_state_trans (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                   struct pal_in4_addr *ing_id,
                                   struct pal_in4_addr *eg_id,
                                   u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Number of times the state has changed

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_st_type

This function gets the storage type of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_st_type (u_int32_t *tn_ix, u_int32_t *inst_ix,
                               struct pal_in4_addr *ing_id,
                               struct pal_in4_addr *eg_id,
                               u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`ix_len`	Index length

Output Parameters

`tn_ix`	Tunnel index of the next tunnel instance
`inst_ix`	Instance index of the next tunnel instance
`ing_id`	Ingress LSR identifier of the next tunnel instance
`eg_id`	Egress LSR identifier of the next tunnel instance
`ret`	This constant from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ST_TYPE_VOLATILE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_next_tn_total_uptime

This function gets the aggregate up time for all instances of the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_total_uptime (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                    struct pal_in4_addr *ing_id,
                                    struct pal_in4_addr *eg_id,
                                    u_int32_t ix_len,u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier

eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Up time

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_next_tn_xc_pointer

This function gets the pointer to a row in mplsXCTable for the next tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_next_tn_xc_pointer (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                  struct pal_in4_addr *ing_id,
                                  struct pal_in4_addr *eg_id,
                                  u_int32_t ix_len,char **ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
ix_len	Index length

Output Parameters

tn_ix	Tunnel index of the next tunnel instance
inst_ix	Instance index of the next tunnel instance
ing_id	Ingress LSR identifier of the next tunnel instance
eg_id	Egress LSR identifier of the next tunnel instance
ret	Pointer to a row in mplsXCTable for the next tunnel

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_active

This function gets the number of active tunnels.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_active (u_int32_t *val)
```

Input Parameters

None

Output Parameters

`val`	Number of active tunnels

Return Value

Always `RSVP_API_GET_SUCCESS`

rsvp_snmp_get_tn_admin_status

This function gets the administration status of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_admin_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                               struct pal_in4_addr *ing_id,
                               struct pal_in4_addr *eg_id,
                               u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	One of these constants from `rsvpd/rsvp_snmp.h` or `rsvpd/rsvp_api.h`:

`RSVP_SNMP_TUNNEL_UP`

`RSVP_SNMP_TUNNEL_DOWN`

`RSVP_TN_ADMIN_DOWN`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_arhop_table_ix

This function gets the index into the mplsTunnelARHopTable entry that specifies the actual hops traversed by this tunnel.

Note: The `ret` parameter is always zero because mplsTunnelARHopTable is not supported.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_arhop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	This parameter is always zero because mplsTunnelARHopTable is not supported

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_chop_table_ix

This function gets the index into the mplsTunnelCHopTable entry that specifies computed hops traversed by this tunnel.

Note: This object is not supported, so the `ret` parameter is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_chop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	This parameter is always zero because mplsTunnelCHopTable is not supported

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_configured

This function gets the number of tunnels configured.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_configured (u_int32_t *val)
```

Input Parameters

None

Output Parameters

`val`	Number of tunnels

Return Value

Always `RSVP_API_GET_SUCCESS`

rsvp_snmp_get_tn_creation_time

This function gets the SysUpTime when the first instance of this tunnel came into existence.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_creation_time (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Time when the first instance of this tunnel came into existence

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_descr

This function gets the description of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_descr (u_int32_t *tn_ix, u_int32_t *inst_ix,
                        struct pal_in4_addr *ing_id,
                        struct pal_in4_addr *eg_id, char **ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Description of the tunnel.

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_excl_any

This function gets this tunnel's exclude-any constraint.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_excl_any (u_int32_t *tn_ix, u_int32_t *inst_ix,
                           struct pal_in4_addr *ing_id,
                           struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	One of these constants from `rsvpd/rsvpd.h`:

`RSVP_TRUE`

`RSVP_FALSE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_hold_prio

This function gets the holding priority for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_hold_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Holding priority

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_hop_ix_next

This function gets the next available hop index.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_hop_ix_next (u_int32_t *ret)
```

Input Parameters

None

Output Parameters

ret	The next available hop index

Return Value

Always RSVP_API_GET_SUCCESS

rsvp_snmp_get_tn_hop_table_ix

This function gets the index into the mplsTunnelHopTable entry that specifies the explicit route hops for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_hop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                               struct pal_in4_addr *ing_id,
                               struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Index into mplsTunnelHopTable

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_if_ix

If mplsTunnelIsIf is true, then this function gets this tunnel's LSR-assigned ifIndex which corresponds to an entry in this interfaces table.

Note: This object is not supported, so `ret` is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_if_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                        struct pal_in4_addr *ing_id,
                        struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	There is no support for this object, so this parameter is always zero

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_incl_all

This function gets this tunnel's include-all constraint.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_incl_all (u_int32_t *tn_ix, u_int32_t *inst_ix,
                           struct pal_in4_addr *ing_id,
                           struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	One of these constants from `rsvpd/rsvpd.h`:
	`RSVP_TRUE`
	`RSVP_FALSE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_incl_any

This function gets this tunnel's include-any constraint.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_incl_any (u_int32_t *tn_ix, u_int32_t *inst_ix,
                           struct pal_in4_addr *ing_id,
                           struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	One of these constants from `rsvpd/rsvpd.h`:
	`RSVP_TRUE`
	`RSVP_FALSE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_inst_prio

This function gets the priority of this tunnel instance within a group of tunnel instances.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_inst_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Priority; one of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_INSTANCE_PRIO_PRIMARY`

`RSVP_INSTANCE_PRIO_SECONDARY`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_inst_uptime

This function gets the total time that the operational status of this tunnel instance has been up.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_inst_uptime (u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id,
                              struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Up time

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_is_if

This function gets whether this tunnel corresponds to an interface represented in the interfaces group table.

Note: This object is not supported, so `ret` is always `RSVP_FALSE`.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_is_if (u_int32_t *tn_ix, u_int32_t *inst_ix,
                        struct pal_in4_addr *ing_id,
                        struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	This constant from `rsvpd/rsvpd.h`:

`RSVP_FALSE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_ix_next

This function gets the next available tunnel index.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_ix_next(u_int32_t *ret)
```

Input Parameters

None

Output Parameters

`ret`	Next available tunnel index

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when there is an internal error

rsvp_snmp_get_tn_last_path_chan

This function gets the time since the last path changed for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_last_path_chan (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Time since the last path changed

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_lopro_inuse

This function gets wether the local repair mechanism is in use for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_lopro_inuse (u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id,
                              struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	One of these constants from `rsvpd/rsvp_snmp.h`:

RSVP_SNMP_FALSE

RSVP_SNMP_TRUE

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_max_hops

This function gets the maximum number of hops that can be specified for a tunnel on this device.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_max_hops (u_int32_t *val)
```

Input Parameters

None

Output Parameters

val	This constant from rsvpd/rsvpd.h:
	RSVP_IP_MULTI_HOP_TTL

Return Value

Always RSVP_API_GET_SUCCESS

rsvp_snmp_get_tn_name

This function gets the canonical name assigned to the tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_name (u_int32_t *tn_ix, u_int32_t *inst_ix,
                       struct pal_in4_addr *ing_id,
                       struct pal_in4_addr *eg_id, char **ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Tunnel name

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_notify_en

This function gets whether mplsTunnelUp, mplsTunnelDown and mplsTunnelRerouted notifications are enabled.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_notify_en (u_int32_t *ret)
```

Input Parameters

None

Output Parameters

ret	One of these constants from rsvpd/rsvpd.h:
RSVP_TRUE	
RSVP_FALSE	

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when there is an internal error

rsvp_snmp_get_tn_notify_max_rate

This function gets the maximum number of notifications per second.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_notify_max_rate (u_int32_t* val)
```

Input Parameters

None

Output Parameters

val	Notification maximum rate

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when there is an internal error

rsvp_snmp_get_tn_oper_status

This function gets the operating status of this tunnel.

Syntax

```
u_int32_t
```

```
rsvp_snmp_get_tn_oper_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id,
                              struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret` One of these constants from `rsvpd/rsvp_snmp.h` or `rsvpd/rsvp_api.h`:

- `RSVP_SNMP_TUNNEL_UP`
- `RSVP_SNMP_TUNNEL_DOWN`
- `RSVP_TN_ADMIN_DOWN`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_owner

This function gets the entity that created this tunnel and is responsible for managing this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_owner (u_int32_t *tn_ix, u_int32_t *inst_ix,
                        struct pal_in4_addr *ing_id,
                        struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret` This constant from `rsvpd/rsvp_snmp.h`:

- `RSVP_SNMP_RSVP_OWNER`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_path_chan

This function gets the number of times the path has changed for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_path_chan (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Number of times the path has changed

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_path_inuse

This function gets the index of the path that was chosen for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_path_inuse (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Index of the path

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_perf_bytes

This function gets the number of bytes forwarded by this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_perf_bytes (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Number of bytes forwarded

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_perf_errors

This function gets the number of packets with errors received by this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_perf_errors (u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id,
                              struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Number of error packets received

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_perf_hc_bytes

This function gets the high capacity counter of number of bytes forwarded by this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_perf_hc_bytes (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id,
                                u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Number of bytes forwarded

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_perf_hc_pkts

This function gets the number of packets forwarded by this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_perf_hc_pkts (u_int32_t *tn_ix, u_int32_t *inst_ix,
                               struct pal_in4_addr *ing_id,
                               struct pal_in4_addr *eg_id,
                               u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Number of packets forwarded

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_perf_pkts

This function gets the number of packets forwarded by this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_perf_pkts (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Number of packets forwarded

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_prim_inst

This function gets the index of the primary instance of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_prim_inst (u_int32_t *tn_ix,
                            u_int32_t *inst_ix, struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Primary instance index

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_prim_timeup

This function gets the total time the primary instance of this tunnel has been active.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_prim_timeup (u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id,
                              struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Active time

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_resource_exburstsize

This function gets the excess burst size in bytes.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_exburstsize (int rs_ix, u_int32_t *ret)
```

Input Parameters

rs_ix	Resource table index

Output Parameters

ret	Excess burst size in bytes

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_resource_frequency

This function gets the resource frequency.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_exburstsize (int rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`ret`	Resource frequency

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_resource_ix_next

This function gets the next available tunnel resource index.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_ix_next(u_int32_t *ret)
```

Input Parameters

None

Output Parameters

`ret`	Next available tunnel resource index

Return Value

Always `RSVP_API_GET_SUCCESS`

rsvp_snmp_get_tn_resource_maxburstsize

This function gets the maximum burst size.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_maxburstsize (int rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix`	Resource table index

Output Parameters

`ret`	Maximum burst size in bytes

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

RSVP_API_GET_ERROR when the resource table is not found

rsvp_snmp_get_tn_resource_maxrate

This function gets the resource maximum rate.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_maxrate (int rs_ix, u_int32_t *ret)
```

Input Parameters

rs_ix	Resource table index

Output Parameters

ret	Resource maximum rate

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the resource table is not found

rsvp_snmp_get_tn_resource_meanburstsize

This function gets the mean burst size in bytes.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_meanburstsize (int rs_ix, u_int32_t *ret)
```

Input Parameters

rs_ix	Resource table index

Output Parameters

ret	Mean burst size in bytes

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the resource table is not found

rsvp_snmp_get_tn_resource_meanrate

This function gets the resource mean rate.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_meanrate (int rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix` Resource table index

Output Parameters

`ret` Resource mean rate

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_tn_resource_rowstatus

This function gets the resource row status.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_rowstatus (int rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix` Resource table index

Output Parameters

`ret` One of these constants from `rsvpd/rsvp_snmp.h`:

- `RSVP_SNMP_ROW_STATUS_ACTIVE`
- `RSVP_SNMP_ROW_STATUS_NOTINSERVICE`
- `RSVP_SNMP_ROW_STATUS_NOTREADY`
- `RSVP_SNMP_ROW_STATUS_CREATEANDGO`
- `RSVP_SNMP_ROW_STATUS_CREATEANDWAIT`
- `RSVP_SNMP_ROW_STATUS_DESTROY`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_tn_resource_storagetype

This function gets the storage type.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_storagetype (int rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix` Resource table index

Output Parameters

`ret` This constant from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ST_TYPE_VOLATILE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_tn_resource_weight

This function gets the resource weight.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_resource_weight (int rs_ix, u_int32_t *ret)
```

Input Parameters

`rs_ix` Resource table index

Output Parameters

`ret` Resource weight

Return Value

`SVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the resource table is not found

rsvp_snmp_get_tn_role

This function gets the role that this tunnel entry/instance represents.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_role (u_int32_t *tn_ix, u_int32_t *inst_ix,
                       struct pal_in4_addr *ing_id,
                       struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix` Tunnel index

`inst_ix` Instance index

`ing_id` Ingress LSR identifier

`eg_id` Egress LSR identifier

Output Parameters

`ret` One of these constants from the `session_role_type` enum in `rsvpd/rsvpd.h`:

`RSVP_SESSION_ROLE_NONE`

RSVP_SESSION_ROLE_INGRESS

RSVP_SESSION_ROLE_TRANSIT

RSVP_SESSION_ROLE_EGRESS

RSVP_SESSION_ROLE_NON_INGRESS

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_row_status

This function gets the row status of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_row_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret — One of these constants from rsvpd/rsvp_snmp.h:

RSVP_SNMP_ROW_STATUS_ACTIVE

RSVP_SNMP_ROW_STATUS_NOTINSERVICE

RSVP_SNMP_ROW_STATUS_NOTREADY

RSVP_SNMP_ROW_STATUS_CREATEANDGO

RSVP_SNMP_ROW_STATUS_CREATEANDWAIT

RSVP_SNMP_ROW_STATUS_DESTROY

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_rs_pointer

This function gets the pointer to the resource table of this tunnel.

Syntax

```
u_int32_t
```

```
rsvp_snmp_get_tn_rs_pointer (u_int32_t *tn_ix,
                             u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id, char **ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Pointer to the resource table

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_sa

This function gets the session attribute bit mask of this tunnel.

Note: This object is not supported, so the `ret` parameter is always zero.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_sa (u_int32_t *tn_ix, u_int32_t *inst_ix,
                     struct pal_in4_addr *ing_id,
                     struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	This object is not supported, so this parameter is always zero

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_setup_prio

This function gets the setup priority of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_setup_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                    struct pal_in4_addr *ing_id,
                                    struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Setup priority

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_sig_proto

This function gets the signaling protocol that was used to set up this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_sig_proto (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	This constant defined in `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_RSVP_PROTO`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_state_trans

This function gets the number of times the state of this tunnel instance has changed.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_state_trans (u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id,
                              struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Number of times the state has changed

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_get_tn_st_type

This function gets the storage type of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_st_type (u_int32_t *tn_ix, u_int32_t *inst_ix,
                          struct pal_in4_addr *ing_id,
                          struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	This constant from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ST_TYPE_VOLATILE`

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_te_dist_proto

This function gets the traffic engineering distribution protocol used by this LSR.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_te_dist_proto (u_int32_t *val)
```

Input Parameters

None

Output Parameters

val	This constant from rsvpd/rsvp_snmp.h: RSVP_SNMP_TE_DIS_OSPF

Return Value

Always RSVP_API_GET_SUCCESS

rsvp_snmp_get_tn_total_uptime

This function gets the aggregate up time for all instances of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_total_uptime (u_int32_t *tn_ix, u_int32_t *inst_ix,
                               struct pal_in4_addr *ing_id,
                               struct pal_in4_addr *eg_id, u_int32_t *ret)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier

Output Parameters

ret	Up time

Return Value

RSVP_API_GET_SUCCESS when the function succeeds

RSVP_API_GET_ERROR when the tunnel instance is not found

rsvp_snmp_get_tn_xc_pointer

This function gets a pointer to a row in mplsXCTable for the tunnel.

Syntax

```
u_int32_t
rsvp_snmp_get_tn_xc_pointer (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id, char **ret)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier

Output Parameters

`ret`	Pointer to a row in mplsXCTable for the tunnel

Return Value

`RSVP_API_GET_SUCCESS` when the function succeeds

`RSVP_API_GET_ERROR` when the tunnel instance is not found

rsvp_snmp_set_hop_addr_un_num

This function sets the interface identifier of the unnumbered interface for this hop.

Note: There is no support for `mplsTunnelHopAddrType unnum` (4) in the backend, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_hop_addr_un_num (u_int32_t *ero_ix, u_int32_t *path_ix,
                               u_int32_t *hop_ix,
                               u_int32_t if_id)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`if_id`	Interface identifier of the unnumbered interface

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_snmp_set_local_protect

This function enables the local repair mechanism for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_local_protect (u_int32_t *tn_ix,
                             u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id,
                             u_char local_protect_flag)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`local_protect_flag`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_FALSE`

`RSVP_SNMP_TRUE`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, `local_protect_flag` is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_admin_status

This function sets the administrative status of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_admin_status(u_int32_t *tn_ix, u_int32_t *inst_ix,
                              struct pal_in4_addr *ing_id, struct pal_in4_addr *eg_id,
                              u_int32_t adminstatus)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`adminstatus`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_TUNNEL_UP`

`RSVP_SNMP_TUNNEL_DOWN`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, the session cannot be started, the tunnel parameters cannot be set or are invalid, `adminstatus` is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_descr

This function sets the description of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_descr (u_int32_t *tn_ix, u_int32_t *inst_ix,
                        struct pal_in4_addr *ing_id,
                        struct pal_in4_addr *eg_id, char *description)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`description`	Description of the tunnel

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, or the the tunnel instance is not found

rsvp_snmp_set_tn_excl_any

This function sets the exclude-any constraint of the tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_excl_any (u_int32_t *tn_ix, u_int32_t *inst_ix,
                           struct pal_in4_addr *ing_id,
                           struct pal_in4_addr *eg_id,
                           u_int32_t exclude_any)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`exclude_any`	One of these constants from `rsvpd/rsvpd.h`:
`RSVP_TRUE`	
`RSVP_FALSE`	

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, the constraint cannot be set, `exclude_any` is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_hold_priority

This function sets the holding priority of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hold_priority (u_int32_t *tn_ix,
                                u_int32_t *inst_ix,
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id,
                                u_int32_t hold_priority)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`hold_priority`	Holding priority

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, `hold_priority` is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_hop_addr_type

This function sets the hop address type.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_addr_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                                u_int32_t *hop_ix, u_int8_t family)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`family`	This constant from `rsvpd/rsvp_api.h`:
`RSVP_IPV4`	

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the tunnel is not active, `family` is not `RSVP_IPV4`, or the hop is not found

rsvp_snmp_set_tn_hop_as_num

This function sets the autonomous system number of the hop.

Note: There is no support for `mplsTunnelHopAddrType asnumber` (3) in the backend, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_as_num (u_int32_t *ero_ix, u_int32_t *path_ix,
                             u_int32_t *hop_ix, u_int32_t hopasnum)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`hopasnum`	Autonomous system number

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_snmp_set_tn_hop_entry_path_comp

This function sets whether CSPF is used to compute this hop.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_entry_path_comp (u_int32_t *ero_ix, u_int32_t *path_ix,
                                      u_int32_t *hop_ix, u_char path_compu)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`path_compu`	One of these constants from `rsvpd/rsvp_snmp.h`:
	`RSVP_SNMP_PATH_COMPUTED`
	`RSVP_SNMP_PATH_NOT_COMPUTED`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the tunnel or hop row status is not active, `path_compu` is not valid, or the hop is not found

rsvp_snmp_set_tn_hop_incl_excl

This function sets whether this hop is included or excluded in the tunnel's path.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_incl_excl (u_int32_t *ero_ix, u_int32_t *path_ix,
                                u_int32_t *hop_ix, u_int32_t hop_include)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`hop_include`	This constant from `rsvpd/rsvp_snmp.h`:
	`RSVP_SNMP_HOP_INCLUDE`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the hop row status is not active, `hop_include` is not valid, or the hop is not found

rsvp_snmp_set_tn_hop_ip_addr

This function sets the hop IP address.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_ip_addr (u_int32_t *ero_ix, u_int32_t *path_ix,
                              u_int32_t *hop_ix, struct pal_in4_addr addr)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`addr`	IP address

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the tunnel is not active or the hop is not found

rsvp_snmp_set_tn_hop_ip_pref_len

This function sets the IP prefix length.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_ip_pref_len (u_int32_t *ero_ix, u_int32_t *path_ix,
                                  u_int32_t *hop_ix, u_int32_t pref_len)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`pref_len`	Prefix length

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the tunnel is not active, `pref_len` is not valid, or the hop is not found

rsvp_snmp_set_tn_hop_lspid

This function sets the LSP identifier of the tunnel for this hop.

Note: There is no support for `mplsTunnelHopAddrType` `lspid` (5) in the backend, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_lspid (u_int32_t *ero_ix, u_int32_t *path_ix,
                            u_int32_t *hop_ix, u_int16_t lspid)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`lspid`	LSP identifier

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_snmp_set_tn_hop_path_set

This function sets the path name.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_path_set (u_int32_t *ero_ix, u_int32_t *path_ix,
                               u_int32_t *hop_ix, char *name)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`name`	Path option name

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the tunnel or hop row status is not active, or the hop is not found

rsvp_snmp_set_tn_hop_row_status

This function sets the row status of the hop table.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_row_status (u_int32_t *ero_ix, u_int32_t *path_ix,
                                 u_int32_t *hop_ix, u_int32_t val)
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`val`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ROW_STATUS_ACTIVE`

`RSVP_SNMP_ROW_STATUS_NOTINSERVICE`

`RSVP_SNMP_ROW_STATUS_CREATEANDWAIT`

`RSVP_SNMP_ROW_STATUS_DESTROY`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when:

- The `val` parameter is not valid
- The hop cannot be found
- Trying to set `RSVP_SNMP_ROW_STATUS_ACTIVE` and:
  - The current status is a value other than `RSVP_SNMP_ROW_STATUS_NOTREADY` or `RSVP_SNMP_ROW_STATUS_NOTINSERVICE`
  - The hop path name flag or IP address flag is not set
  - The path is not MPLS
  - The path cannot be added
- Trying to set `RSVP_SNMP_ROW_STATUS_NOTINSERVICE`:
  - The tunnel is not active
  - The current status is `RSVP_SNMP_ROW_STATUS_NOTREADY` or `RSVP_SNMP_ROW_STATUS_NOTINSERVICE`
- Trying to set `RSVP_SNMP_ROW_STATUS_CREATEANDWAIT` and the hop already exists
- Trying to set `RSVP_SNMP_ROW_STATUS_DESTROY` and the hop path cannot be found

RSVP_FAILURE when memory allocation fails

rsvp_snmp_set_tn_hop_st_type

This function sets the storage type.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_st_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                              u_int32_t *hop_ix, u_int32_t storage_type)
```

Input Parameters

ero_ix	Primary index identifying an explicit route object
path_ix	Secondary index identifying a group of hops for a configured path
hop_ix	Secondary index identifying a particular hop
storage_type	This constant from rsvpd/rsvp_snmp.h: RSVP_SNMP_ST_TYPE_VOLATILE

Output Parameters

None

Return Value

RSVP_API_SET_SUCCESS when the function succeeds

RSVP_API_SET_ERROR when the hop not found

rsvp_snmp_set_tn_hop_table_ix

This function sets the index into the mplsTunnelHopTable entry that specifies the explicit route hops for this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_table_ix (u_int32_t *tn_ix, u_int32_t *inst_ix,
                               struct pal_in4_addr *ing_id,
                               struct pal_in4_addr *eg_id, u_int16_t id)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
id	Index into mplsTunnelHopTable

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, the path is not found, or the tunnel is not found

rsvp_snmp_set_tn_hop_type

This function sets the hop type which denotes whether this tunnel hop is routed in a strict or loose fashion.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_hop_type (u_int32_t *ero_ix, u_int32_t *path_ix,
                           u_int32_t *hop_ix, u_int32_t is_strict))
```

Input Parameters

`ero_ix`	Primary index identifying an explicit route object
`path_ix`	Secondary index identifying a group of hops for a configured path
`hop_ix`	Secondary index identifying a particular hop
`is_strict`	One of these constants from `rsvpd/rsvp_api.h`:
	`RSVP_HOP_TYPE_IS_STRICT`
	`RSVP_HOP_TYPE_IS_LOOSE`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the tunnel is not active, `is_strict` is not valid, or the hop is not found

rsvp_snmp_set_tn_incl_all

This function sets this tunnel's include-all constraint.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_incl_all (u_int32_t *tn_ix, u_int32_t *inst_ix,
                           struct pal_in4_addr *ing_id, struct pal_in4_addr *eg_id,
                           u_int32_t include_all)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`include_all`	One of these constants from `rsvpd/rsvpd.h`:

RSVP_TRUE

RSVP_FALSE

Output Parameters

None

Return Value

RSVP_API_SET_SUCCESS when the function succeeds

RSVP_API_SET_ERROR when the trunk owner is not SNMP, the session role is not ingress, the row status is not active, the constraint cannot be set, include_all is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_incl_any

This function sets this tunnel's include-any constraint.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_incl_any (u_int32_t *tn_ix, u_int32_t *inst_ix,
                           struct pal_in4_addr *ing_id,
                           struct pal_in4_addr *eg_id,
                           u_int32_t include_any)
```

Input Parameters

tn_ix	Tunnel index
inst_ix	Instance index
ing_id	Ingress LSR identifier
eg_id	Egress LSR identifier
include_any	One of these constants from rsvpd/rsvpd.h:
RSVP_TRUE	
RSVP_FALSE	

Output Parameters

None

Return Value

RSVP_API_SET_SUCCESS when the function succeeds

RSVP_API_SET_ERROR when the trunk owner is not SNMP, the session role is not ingress, the row status is not active, the constraint cannot be set, include_any is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_inst_prio

This function sets the tunnel instance priority.

Syntax

```
u_int32_t
```

```
rsvp_snmp_set_tn_inst_prio (u_int32_t *tn_ix, u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id, u_int32_t prio)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`prio`	Priority; one of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_INSTANCE_PRIO_PRIMARY`
`RSVP_INSTANCE_PRIO_SECONDARY`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, the row status is not active, `prio` is not a valid value, or the the tunnel instance is not found

rsvp_snmp_set_tn_is_if

This function sets whether this tunnel corresponds to an interface represented in the interfaces group table.

Note: There is no support for this object, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_is_if (u_int32_t *tn_ix, u_int32_t *inst_ix,
                        struct pal_in4_addr *ing_id,
                        struct pal_in4_addr *eg_id, int interface)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`interface`	Interface identifier

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_snmp_set_tn_name

This function assigns a canonical name to the tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_name (u_int32_t *tn_ix, u_int32_t *inst_ix,
                       struct pal_in4_addr *ing_id,
                       struct pal_in4_addr *eg_id, char *name)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`name`	Tunnel name

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, the row status is not active, memory for `name` cannot be allocated, or the the tunnel instance is not found

rsvp_snmp_set_tn_notify_en

This function enables or disables mplsTunnelUp, mplsTunnelDown and mplsTunnelRerouted notifications.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_notify_en (u_int32_t val)
```

Input Parameters

`val`	One of these constants from `rsvpd/rsvpd.h`:
	`RSVP_TRUE`
	`RSVP_FALSE`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the RSVP master is not found

rsvp_snmp_set_tn_notify_max_rate

This function sets the maximum number of notifications per second.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_notify_max_rate (u_int32_t val)
```

Input Parameters

`val`	Notification maximum rate

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the RSVP master is not found

rsvp_snmp_set_tn_path_inuse

This function sets the index of the path that was chosen for this tunnel.

Note: This object is not supported, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_path_inuse (u_int32_t *tn_ix,
                             u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id,
                             u_int32_t path_id)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`path_id`	Path identifier

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_snmp_set_tn_resource_exburstsize

This function sets the excess burst size in bytes.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_exburstsize (int rs_ix, u_int32_t exburstsize)
```

Input Parameters

`rs_ix`	Resource table index
`exburstsize`	Excess burst size in bytes

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row status is active, or the tunnel is active

rsvp_snmp_set_tn_resource_frequency

This function sets the resource frequency.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_frequency (int rs_ix, u_int32_t frequency)
```

Input Parameters

`rs_ix`	Resource table index
`frequency`	One of these constants from `rsvpd/rsvp_api.h`:

`RSVP_FREQUENCY_UNSPECIFIED`

`RSVP_FREQUECNY_FREQUENT`

`RSVP_FREQUENCY_VERYFREQUENT`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row status is active, the tunnel is active, or `frequency` is an invalid value

rsvp_snmp_set_tn_resource_maxburstsize

This function sets the maximum burst size in bytes.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_maxburstsize (int rs_ix, u_int32_t maxbursrsize)
```

Input Parameters

`rs_ix`	Resource table index
`maxbursrsize`	Maximum burst size in bytes

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row status is active, or the tunnel is active

rsvp_snmp_set_tn_resource_maxrate

This function sets the resource maximum rate.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_maxrate (int rs_ix, u_int32_t maxrate)
```

Input Parameters

`rs_ix`	Resource table index
`maxrate`	Resource maximum rate

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row status is active, or the tunnel is active

rsvp_snmp_set_tn_resource_meanburstsize

This function sets the mean burst size in bytes.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_meanburstsize (int rs_ix, u_int32_t meanburstsize)
```

Input Parameters

`rs_ix`	Resource table index
`meanburstsize`	Mean burst size in bytes

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row status is active, or the tunnel is active

rsvp_snmp_set_tn_resource_meanrate

This function sets the resource mean rate.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_meanrate (int rs_ix, u_int32_t meanrate)
```

Input Parameters

`rs_ix`	Resource table index
`meanrate`	Resource mean rate

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row status is active, or the tunnel is active

rsvp_snmp_set_tn_resource_rowstatus

This function sets the row status.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_rowstatus(int rs_ix, u_int32_t val)
```

Input Parameters

`rs_ix`	Resource table index
`val`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ROW_STATUS_ACTIVE`

`RSVP_SNMP_ROW_STATUS_NOTINSERVICE`

`RSVP_SNMP_ROW_STATUS_CREATEANDGO`

`RSVP_SNMP_ROW_STATUS_CREATEANDWAIT`

`RSVP_SNMP_ROW_STATUS_DESTROY`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when:

- The `val` parameter is an invalid value
- Trying to set `RSVP_SNMP_ROW_STATUS_ACTIVE` and the resource is not found or the tunnel is active
- Trying to set `RSVP_SNMP_ROW_STATUS_CREATEANDGO` and the resource already exists or cannot be created
- Trying to set `RSVP_SNMP_ROW_STATUS_CREATEANDWAIT` and the resource already exists or cannot be created
- Trying to set `RSVP_SNMP_ROW_STATUS_NOTINSERVICE` and the resource is not found, the tunnel is active, or the row status is `RSVP_SNMP_ROW_STATUS_NOTREADY` or `RSVP_SNMP_ROW_STATUS_NOTINSERVICE`
- Trying to set `RSVP_SNMP_ROW_STATUS_DESTROY` and the resource is not found or the tunnel is active

rsvp_snmp_set_tn_resource_storagetype

This function sets the storage type.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_storagetype (int rs_ix, u_int32_t storagetype)
```

Input Parameters

`rs_ix`	Resource table index
`storagetype`	This constant from `rsvpd/rsvp_snmp.h`:
	`RSVP_SNMP_ST_TYPE_VOLATILE`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row staus is not active, the tunnel is active, or `storagetype` is an invalid value

rsvp_snmp_set_tn_resource_weight

This function sets the resource weight.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_resource_weight (int rs_ix, u_int32_t resourceweight)
```

Input Parameters

`rs_ix`	Resource table index

`resourceweight` Resource weight

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the resource is not found, the row staus is not active, the tunnel is active, or `resourceweight` is an invalid value

rsvp_snmp_set_tn_role

This function sets the role that this tunnel entry/instance represents.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_role (u_int32_t *tn_ix, u_int32_t *inst_ix,
                       struct pal_in4_addr *ing_id,
                       struct pal_in4_addr *eg_id, u_int32_t role)
```

Input Parameters

`tn_ix` Tunnel index

`inst_ix` Instance index

`ing_id` Ingress LSR identifier

`eg_id` Egress LSR identifier

`role` This constant from the `session_role_type` enum in `rsvpd/rsvpd.h`:

`RSVP_SNMP_TN_ROLE_INGRESS`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, `role` is not a valid value, or the tunnel instance is not found

rsvp_snmp_set_tn_row_status

This function sets the row status of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_row_status (u_int32_t *tn_ix, u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t row_status)
```


Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`row_status`	One of these constants from `rsvpd/rsvp_snmp.h`:

`RSVP_SNMP_ROW_STATUS_ACTIVE`

`RSVP_SNMP_ROW_STATUS_NOTINSERVICE`

`RSVP_SNMP_ROW_STATUS_NOTREADY`

`RSVP_SNMP_ROW_STATUS_CREATEANDWAIT`

`RSVP_SNMP_ROW_STATUS_DESTROY`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when:

- The trunk owner is not SNMP
- The session role is not ingress
- The `row_status` parameter is not a valid value
- Trying to set `RSVP_SNMP_ROW_STATUS_ACTIVE` and:
  - The row status is not `RSVP_SNMP_ROW_STATUS_NOTINSERVICE`
  - The tunnel instance is not found
  - The tunnel parameters cannot be set
  - The RSVP session is not found
- Trying to set `RSVP_SNMP_ROW_STATUS_CREATEANDWAIT` and:
  - The tunnel instance is not found
  - The RSVP session was found
- Trying to set `RSVP_SNMP_ROW_STATUS_NOTINSERVICE` and the RSVP session is not found
- Trying to set `RSVP_SNMP_ROW_STATUS_DESTROY` and:
  - The trunk is not found
  - The tunnel instance is not found

rsvp_snmp_set_tn_rs_pointer

This function sets the pointer to the resource table of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_rs_pointer (u_int32_t *tn_ix, u_int32_t *inst_ix,
```

```
                                struct pal_in4_addr *ing_id,
                                struct pal_in4_addr *eg_id, int index)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`index`	Pointer to resource table

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when:

- The trunk owner is not SNMP
- The session role is not ingress
- The resource table is not found
- The mean rate (used to initialize the bandwidth) is not found
- The tunnel instance is not found

rsvp_snmp_set_tn_sa

This function sets the session attributes of the tunnel.

Note: This object is not supported, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_sa (u_int32_t *tn_ix,
                     u_int32_t *inst_ix, struct pal_in4_addr *ing_id,
                     struct pal_in4_addr *eg_id, u_char flags)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`flags`	Session attributes

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_snmp_set_tn_setup_priority

This function sets the setup priority of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_setup_priority (u_int32_t *tn_ix,
                                 u_int32_t *inst_ix,
                                 struct pal_in4_addr *ing_id,
                                 struct pal_in4_addr *eg_id,
                                 u_int32_t setup_priority)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`setup_priority`	Setup priority

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when:

- The trunk owner is not SNMP
- The session role is not ingress
- The `setup_priority` parameter is not a valid value
- The tunnel instance is not found

rsvp_snmp_set_tn_sig_proto

This function sets the signaling protocol used to set up the tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_sig_proto (u_int32_t *tn_ix,
                            u_int32_t *inst_ix,
                            struct pal_in4_addr *ing_id,
                            struct pal_in4_addr *eg_id,
                            u_int32_t protocol_type)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`protocol_type`	This constant defined in `rsvpd/rsvp_snmp.h`: `RSVP_SNMP_RSVP_PROTO`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, `protocol_type` is not a valid value, or the tunnel instance is not found

rsvp_snmp_set_tn_st_type

This function sets the storage type of this tunnel.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_st_type (u_int32_t *tn_ix, u_int32_t *inst_ix,
                          struct pal_in4_addr *ing_id,
                          struct pal_in4_addr *eg_id,
                          u_int32_t storage_type)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`storage_type`	This constant from `rsvpd/rsvp_snmp.h`: `RSVP_SNMP_ST_TYPE_VOLATILE`

Output Parameters

None

Return Value

`RSVP_API_SET_SUCCESS` when the function succeeds

`RSVP_API_SET_ERROR` when the trunk owner is not SNMP, the session role is not ingress, `storage_type` is not a valid value, or the RSVP session is not found

rsvp_snmp_set_tn_xc_pointer

This function sets a pointer to a row in mplsXCTable for the tunnel.

Note: This object is not supported, so this function always returns `RSVP_API_SET_ERROR`.

Syntax

```
u_int32_t
rsvp_snmp_set_tn_xc_pointer (u_int32_t *tn_ix, u_int32_t *inst_ix,
                             struct pal_in4_addr *ing_id,
                             struct pal_in4_addr *eg_id, oid *xc_ptr)
```

Input Parameters

`tn_ix`	Tunnel index
`inst_ix`	Instance index
`ing_id`	Ingress LSR identifier
`eg_id`	Egress LSR identifier
`xc_ptr`	Pointer to a row in mplsXCTable for the tunnel

Output Parameters

None

Return Value

Always `RSVP_API_SET_ERROR`

rsvp_tn_down

This function is called for an mplsTunnelDown notification.

Syntax

```
void
rsvp_tn_down (struct rsvp_session *session)
```

Input Parameters

`session`	RSVP session

Output Parameters

None

Return Value

None

rsvp_tn_reroute

This function is called for an mplsTunnelRerouted notification.

Syntax

```
void
rsvp_tn_reroute (struct rsvp_session *session)
```

Input Parameters

`session`	RSVP session

Output Parameters

None

Return Value

None

rsvp_tn_up

This function is called for an mplsTunnelUp notification.

Syntax

```
void
rsvp_tn_up (struct rsvp_session *session))
```

Input Parameters

`session`	RSVP session

Output Parameters

None

Return Value

None

CHAPTER 15 Point-to-Multipoint Services

Standard MPLS traffic engineering (MPLS-TE) supports strict QoS guarantees, resource optimization, and fast failure recovery, but it is limited to point-to-point (P2P) label switched paths (LSP). Many service providers use native IP multicast in order to provide IP TV broadcasting, multicast content delivery and other services. However, IP multicast has issues that limit its deployment at the core, including:

- Lack of traffic engineering capabilities
- Requires a few seconds or more to re-converge after failures
- Requires maintenance of a large number states of across the core, making it unwieldy when scaling

Without multicast capabilities in the core network, there is a need to support point-to-multipoint (P2MP) services using traffic-engineered LSPs in order to deliver data from a single source to one or more receivers. A P2MP traffic-engineered LSP is a TE LSP with a single ingress LSR (label switched route), one or more egress LSRs, and is unidirectional.

Overview

In addition to traversing standard transit LSRs, which connect a single in-segment to a single out-segment, P2MP LSPs also traverse Branch LSRs, which connect a single in-segment to multiple out-segments, and bud LSRs which connect to one or more out-segments, thereby permitting P2MP operation.

P2MP services may be supported with any combination of P2P and P2MP LSPs, depending on the degree of optimization required within the network, and these LSPs may be traffic-engineered. Multipoint to-multipoint (MP2MP) services, which deliver data from more than one source to one or more receivers, can also be supported with a combination of P2P and P2MP LSPs.

P2MP tunnel processing in RSVP-TE used the same backend structure and trunk table as does P2P processing. It shares the same name space for tunnel name configurations, and stores LSP data belonging to a session using the main constructs as those used by P2P.

Architecture

For P2MP LSP processing, the transport path hierarchy is:

1. P2MP tunnel
2. P2MP LSP
3. Source-to-Leaf (S2L) Sub-LSP

Point-to-Multipoint Tunnels

Unlike P2P tunnels, P2MP tunnels do not contain egress information. Instead, egress information is a property of each source-to-leaf (S2L) sub-LSP. Each ingress node also generates a unique P2MP ID for each P2MP trunk. P2MP tunnels are indexed by the tuple (Ingress, Trunk ID, P2MP ID) in the global trunk table.

The reservation style used for P2P and stored per-LSP is a per-tunnel property in P2MP. Two structures, the main session list and the MBB (make-before-break) session list store data associated with P2MP tunnels..

P2MP tunnels share their name space and identifier space with P2P tunnels. The Ingress, Transit Upstream, Transit Downstream, and Egress FSM used for signaling P2MP LSP, are the same as those used for P2P. P2MP utilizes the same interfaces between RSVP-TE and CSPF, RSVP-TE and NSM QoS, and RSVP-TE and the Label Manager as P2P LSP.

The operational status of a P2MP tunnel requires that at least one of its LSPs is up, in order to declare itself as operationally UP.

Point-to-Multipoint Label Switched Paths

A placeholder that represents P2MP LSP is defined (struct that contains all configuration information except egress, the path computation scheme and any explicit hop for a destination. Configuration information is passed to individual S2L sessions when configured within an P2MP LSP.

A P2MP LSP is only created when a primary or secondary LSP within a P2MP tunnel is established. Each P2MP LSP is assigned a unique LSP ID from the LSP ID table, and this resource is shared with other P2MP LSPs. P2MP LSP configurations differ from P2P LSP in that they are hierarchical. P2MP LSPs have a view of their S2L sub-LSPs that allow them to signal their sub-LSPs to he same destination with different sub-group IDs.

The operational status of a P2MP LSP depends on the operational status of all component sub-LSPs; all must be UP in order for the LSP can declare itself to be operationally UP.

Source-to-Leaf Sub-LSP

A source-to-leaf (S2L) sub-LSP is a component that identifies each source-to-leaf path in a P2MP LSP. An S2L Sub-LSP is represented by the RSVP session structure (struct rsvp_session). Under normal circumstances, one S2L Sub-objects exists for each destination within a P2MP LSP. An S2L sub-LSP is explicitly created within a P2MP LSP when a new S2L destination is configured with the desired properties. A sub-LSP explicitly holds configuration of egress, path computation scheme and explicit hops, and derives all other TE properties from the parent P2MP LSP when it is created. P2MP S2L sub-LSP support incremental update of their properties with MBB processing.

Every S2L sub-LSP requires a unique, system-wide identifier for propagation in the P2MP_SENDER_TEMPLATE object. The identifiers are generated at the ingress node of every P2MP LSP using a sub-group identifier that is stored in the LSP ID field of the RSVP session structure.

The operational status of an S2L sub-LSP depends on its signaling state.

Figure 15-1 depicts a comparison of an RSVP P2P LSP and an RSVP P2MP S2L LSP.

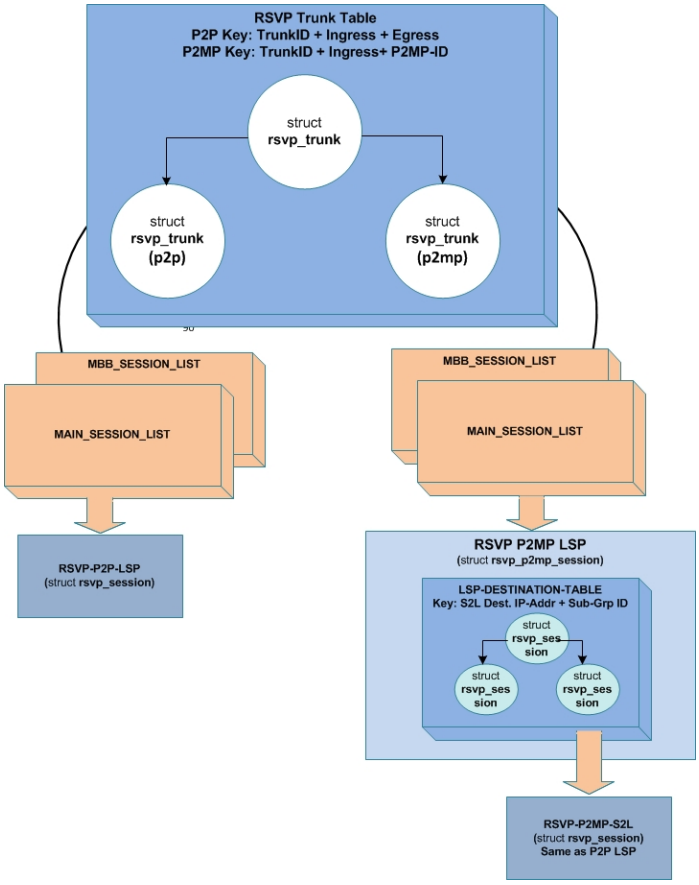

Figure 15-1: P2P LSP Compared to P2MP LSP

Explicit Route Object Computation

The Constrained Shortest Path First (CSPF) module, part of the ZebOS-XP Open Shortest Path First (OSPF) module is used by RSVP-TE to dynamically compute the constraint-based path for each

- P2MP sub-LSP that belongs to a P2MP LSP
- P2MP sub-LSP whose parent P2MP LSP is undergoing make-before-break (MBB)
- P2MP sub-LSP undergoing incremental update.

Messages between RSVP-TE and CSPF are used request optimum path computation. To identify an S2L sub-LSP requesting CSPF service, a P2MP ID and a S2L Sub-Group ID are implemented.

Quality of Service Reservation Request

The RSVP-TE standard requires that every LSP request undergoes two cycles on each node before securing a guaranteed QoS for its flow. The first is the probe cycle, and the second is the reservation cycle. Both cycles are used with every S2L sub-LSP for P2MP LSP. Since bandwidth requests are per-LSP properties, all S2L sub-LSPs can share a QoS resources if the also share the same downstream interface.

Before issuing a Probe or Reserve request to the QoS manager in NSM, a determination is made as to whether a new Sub-LSP shares the downstream interface with any other Sub-LSP that belongs to the same P2MP LSP. If it does, RSVP-TE suppresses the Probe and Reserve messages and the QoS resource is copied to the session. Therefore,

Probe and Reserve messages. As a consequence, Probe and Reserve messages are originated for the first signaled sub-LSP within a P2MP LSP.

LSP Property updates

Two kinds of updates are possible for a P2MP LSP:

- LSP Level Properties
- Sub-LSP Level Properties

LSP Level Properties

Once configured, the following properties of an LSP can be updated on the ingress:

- Bandwidth
- TE Class
- Administrative groups
- Setup and Hold priorities
- Hop-limit
- Route and/or label record
- Fast reroute properties

This should result in make-before-break style signaling of component Sub-LSPs on the ingress. On the lines of P2P, MBB sibling is cloned from the existing P2MP LSP and updated properties are recorded in this cloned LSP. There are two possibilities:

1. Original LSP is Steady State, that is, the MBB LSP is cloned from the original LSP itself.
2. Another MBB, for example, Clone1, is in progress for the original LSP, which means that the MBB LSP 1 can be created from Clone 1, instead of the original LSP, or that Clone 1 is released.

A copy of all the component sub-LSPs that are cloned from the parent LSP to this MBB LSP. The sub-group ID is retained for all the sub LSPs. Again, there are two possibilities:

1. Sub-LSP of original LSP in steady state, that is it is cloned from the original session and moved to the MBB LSP's placeholder
2. Sub-LSP of original LSP is undergoing incremental update, which means that out of the two Sub-LSPs within the Original LSP, the copy representing the incremental update is chosen to be cloned within the MBB LSP. An incremental update to the sub-LSP of the original LSP is released.

Signaling is subsequently sequentially initiated for the component sub-LSPs of the MBB. Data management of the original sibling takes place only at the LSP level. This means that the S2L sub-LSPs of the MBB sibling need not keep track of their counterpart S2L sub-LSPs belonging to the original session. MBB is complete only when any of the following is true:

- All component sub-LSPs of the MBB LSP which are in UP state in their counterpart have been successfully signaled.
- A counterpart session is cleared, thereby impacting its S2L sub-LSPs.

Sub-LSP Level Properties

The following properties of a Sub-LSP can be updated at the ingress:

- Path computation type (CSPF/non-CSPF),
- Explicit hop

This results in creation of another Sub-LSP (referred to as the “Update-Sub-LSP”) to the same destination. The clone-sub-LSP is re-signaled with updated properties set in make-before-break fashion. The parent P2MP LSP and other Sub-LSPs belonging to the LSP should remain un affected by this.

There are two possibilities here based on the current state of the Sub-LSP’s parent P2MP LSP:

1. Parent LSP is in Steady State, meaning that the Update-Sub-LSP is cloned from of the Sub-LSP from Parent LSP.
2. MBB is in progress for the Parent LSP. In this case, the Update-Sub-LSP is cloned from the Sub-LPS of the MBB LSP. an older copy of the sub-LPS within the MBB LSP is released. Release refers to:
   - Clearance of sub-LSP state on all nodes by means of PATH Tear triggered from Ingress
   - Resulting clearance of context and memory allocated to this Sub-LSP on all nodes
   - This update-Sub-LSP is integrated into the MBB sibling’s sub-LSP. From this point on, incremental updates cease to exist for this S2L and be implicitly handled by the MBB.

There are two more possibilities depending on whether another incremental update is in progress for a Sub-LSP:-

1. Original Sub LSP of original LSP in steady state, that is, an Update-Sub-LSP is cloned from this sub-LSP
2. The original Sub-LSP, representing the incremental update, is chosen to be cloned within it as an Update-Sub-LSP. Existing incremental update sub-LSP is released.
3. Signaling is subsequently initiated for the Update-sub-LSP. Management of the original and update Sub-LSPs takes place at the LSP level and also at the Update-Sub-LSP level. Incremental update is complete only when the Update-Sub-LSP has been successfully signaled. At this point the original Sub-LSP is released from the corresponding P2MP LSP. In addition, the update-Sub-LSP now becomes the main Sub-LSP for this destination within the P2MP LSP.

Re-merge Detection

Re-merge of a P2MP LSP is defined as An ingress or transit node that creates a branch of a P2MP LSP, a re-merge branch, that intersects the P2MP LSP at another node farther down the tree. The diagram illustrates a re-merge scenario:

Figure 15-2: P2MP LSP Re-merge

At Ingress

Re-merge detection takes place at the Ingress, when an a S2L Path computation uses CSPF, the route calculation has succeeded and it is received by RSVP-TE. The Path Computed by CSPF for the new S2L LSP needs to be cross-verified with EROs of other sessions belonging to the same P2MP LSP. When all EROs match again downstream of the Branch point, then re-merge has taken place.

On Non-Ingress Nodes

In the transit and egress nodes, if path messages (each corresponding to different S2L LSP of the same P2MP LSP) are received from different incoming interfaces, the node considers that re-merge has taken place.

Fast Reroute Support

For P2MP tunnels, Fast Reroute (FRR) is supported as follows.

- FRR support for P2MP Tunnels is limited to the bypass scheme only.
- FRR scope is limited to the selection of an appropriate bypass tunnel from RSVP-TE side.
- Primary Tunnel mapping to FRR bypass tunnel information is not transmitted to NSM; this is also true for P2MP tunnel mapping. As a result, no switchover mechanism can be demonstrated for P2MP.
- FRR properties configured at the LSP level are only applicable to the primary LSP.
- Topology needs to be over-provisioned to ensure all the S2L Sub-LSP have back-up tunnels.

Command API

The functions in this section are called by the commands used to configure RSVP-TE P2MP LSP. These commands are defined in the *RSVP-TE Command Reference*.

rsvp_api_delete_p2mp_trunk

This call deletes a configured P2MP trunk.

Syntax

```
void
rsvp_api_delete_p2mp_trunk (struct rsvp_trunk *trunk)
```

Input Parameters

`trunk`	P2MP trunk to delete

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_admin_group_set

This call sets Administrative Group attributes for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_admin_group_set (struct rsvp_p2mp_session *session,
                                       u_char type, char *name)
```


Input Parameters

`session`	P2MP session
`type`	Session type, either: RSVP_SESSION_AFFINITY_EXCLUDE_ANY RSVP_SESSION_AFFINITY_INCLUDE_ANY
`name`	Administrative group name

Output Parameters

None

Return Value

RSVP_ERR_INVALID_ARGS when Invalid arguments are passed

RSVP_ERR_INTERNAL when there is a internal error

RSVP_SUCCESS when the call is successful

RSVP_FAILURE when the call is fails

rsvp_api_p2mp_session_admin_group_unset

This call removes configured administrative group attributes for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_admin_group_unset (struct rsvp_p2mp_session *session,
                                         u_char type, char *name)
```

Input Parameters

`orig_session`	P2MP session
`type`	Session type, either: RSVP_SESSION_AFFINITY_EXCLUDE_ANY RSVP_SESSION_AFFINITY_INCLUDE_ANY
`name`	Administrative group name

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_disable_affinity_packing

This call disables pack affinity for a P2MP session.

Syntax

void

```
rsvp_api_p2mp_session_disable_affinity_packing (struct rsvp_p2mp_session
                                                *orig_session)
```

Input Parameters

`orig_session` P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_disable_label_record

This call disables label recording for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_disable_label_record (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session` P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_disable_route_record

This call disables route recording for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_disable_route_record (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session` P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_enable_affinity_packing

This call enables affinity packing for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_enable_affinity_packing (struct rsvp_p2mp_session
                                               *orig_session)
```

Input Parameters

orig_session	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_enable_label_record

This call enables label recording.

Syntax

```
void
rsvp_api_p2mp_session_enable_label_record (struct rsvp_p2mp_session
                                           *orig_session)
```

Input Parameters

orig_session	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_enable_route_record

This call enables route recording (RRO).

Syntax

```
void
rsvp_api_p2mp_session_enable_route_record (struct rsvp_p2mp_session *orig_session))
```

Input Parameters

orig_session	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_free

This call deletes a configured P2MP LSP.

Syntax

```
void
rsvp_api_p2mp_session_free (struct rsvp_p2mp_session *p2mp_session)
```

Input Parameters

`p2mp_session`	P2MP LSP session to delete

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_frr_bandwidth_set

This call sets the bandwidth for FRR LSP.

Note: This takes effect only when protection is enabled.

Syntax

```
int
rsvp_api_p2mp_session_frr_bandwidth_set (struct rsvp_p2mp_session *session,
                                         float32_t bandwidth)
```

Input Parameters

`session`	P2MP session
`bandwidth`	Bandwidth required for backup LSP

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_frr_bandwidth_unset

This call removes configured bandwidth for the backup LSP.

Syntax

```
int
rsvp_api_p2mp_session_frr_bandwidth_unset (struct rsvp_p2mp_session *session)
```

Input Parameters

`session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_frr_node_protection_set

This call enables node protection.

Note: This is available only when protection is enabled.

Syntax

```
int
rsvp_api_p2mp_session_frr_node_protection_set (struct rsvp_p2mp_session *session)
```

Input Parameters

`session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_frr_node_protection_unset

This call disables node protection.

Note: This is available only when protection is enabled.

Syntax

```
int
rsvp_api_p2mp_session_frr_node_protection_unset  (struct rsvp_p2mp_session *session)
```

Input Parameters

`session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_frr_protection_set

This call enables Fast Reroute (FRR) facility backup.

Syntax

```
int
rsvp_api_p2mp_session_frr_protection_set (struct rsvp_p2mp_session *session,
                                          protection_t protection)
```

Input Parameters

`session`	P2MP session
`protection`	RSVP_FRR_PROTECTION_FACILITY

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_frr_protection_unset

This call disables FRR for P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_frr_protection_unset  (struct rsvp_p2mp_session *session)
```

Input Parameters

`session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_set_bandwidth

This call sets a bandwidth value for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_set_bandwidth (struct rsvp_p2mp_session *orig_session,
                                     float32_t bandwidth)
```

Input Parameters

`orig_session`	P2MP session
`bandwidth`	Bandwidth (Float Value)

Output Parameters

None

Return Value

None.

rsvp_api_p2mp_session_set_hold_priority

This call sets a hold priority for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_set_hold_priority (struct rsvp_p2mp_session *orig_session,
                                         u_char val)
```

Input Parameters

`orig_session`	P2MP session
`val`	Hold priority

Output Parameters

None

Return Value

RSVP_FAILURE when the call fails

RSVP_SUCCESS when the call is successful

rsvp_api_p2mp_session_set_hop_limit

This call sets the maximum number of hops that an LSP can traverse.

Syntax

```
void
rsvp_api_p2mp_session_set_hop_limit (struct rsvp_p2mp_session *session, u_char val)
```

Input Parameters

`session`	P2MP session
`val`	Hop limit

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_set_retry_count

This call sets a retry count for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_set_retry_count (struct rsvp_p2mp_session *session,
                                       u_int16_t val)
```

Input Parameters

`session`	P2MP session
`val`	Retry count value

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_set_retry_interval

This call sets a retry interval for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_set_retry_interval (struct rsvp_p2mp_session *session,
                                          u_int16_t val)
```

Input Parameters

`session`	P2MP session
`val`	Retry timer interval value

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_set_setup_priority

This call sets a setup priority for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_set_setup_priority (struct rsvp_p2mp_session
                                          *orig_session, u_char val)
```

Input Parameters

`orig_session`	P2MP session
`val`	Setup priority value

Output Parameters

None

Return Value

RSVP_FAILURE when the call fails

RSVP_SUCCESS when the call is successful

rsvp_api_p2mp_session_set_traffic

This call sets a traffic type for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_set_traffic (struct rsvp_p2mp_session *orig_session,
                                   u_char traffic)
```

Input Parameters

`orig_session`	P2MP session
`traffic`	Traffic type, one of: QOS_TRAFFIC_TYPE_GUARANTEED QOS_TRAFFIC_TYPE_CONTROLLED

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_unset_bandwidth

This call removes the configured bandwidth for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_unset_bandwidth  (struct rsvp_p2mp_session
                                        *orig_session)
```

Input Parameters

`orig_session` P2MP session

Output Parameters

None

Return Value

RSVP_FAILURE if the call fails

RSVP_SUCCESS if the call is successful

rsvp_api_p2mp_session_unset_hold_priority

This call removes a configured hold priority for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_unset_hold_priority (struct rsvp_p2mp_session
                                           *orig_session)
```

Input Parameters

`orig_session` P2MP session

Output Parameters

None

Return Value

RSVP_FAILURE if the call fails

RSVP_SUCCESS if the call is successful

rsvp_api_p2mp_session_unset_hop_limit

This call removes a configured hop limit for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_unset_hop_limit (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session` P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_unset_retry_count

This call removes a configured retry count for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_unset_retry_count (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_unset_retry_interval

This call removes a configured retry interval for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_unset_retry_interval (struct rsvp_p2mp_session
                                            *orig_session)
```

Input Parameters

`orig_session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_p2mp_session_unset_setup_priority

This call removes a configure setup priority for a P2MP session.

Syntax

```
int
rsvp_api_p2mp_session_unset_setup_priority (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session`	P2MP session

Output Parameters

None

Return Value

RSVP_FAILURE when the call fails

RSVP_SUCCESS when the call is successful

rsvp_api_p2mp_session_unset_traffic

This call removes a configured traffic type for a P2MP session.

Syntax

```
void
rsvp_api_p2mp_session_unset_traffic  (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_api_s2l_session_free

This call deletes a P2MP S2L destination.

Syntax

```
void
rsvp_api_s2l_session_free (struct rsvp_session *session)
```

Input Parameters

`session`	P2MP S2L session to delete

Output Parameters

None

Return Value

None

rsvp_api_s2l_session_param_set

This call sets S2L session parameters and triggers the FSM.

Syntax

```
void
rsvp_api_s2l_session_param_set (struct rsvp_session *orig_session,
                                struct rsvp_path *path,
                                bool_t enable_cspf)
```

Input Parameters

`orig_session`	S2L Session
`path`	Explicit path (ERO)
`enable_cspf`	One of:
	RSVP_TRUE to enable CSPF
	RSVP_FALSE to disable CSPF

Output Parameters

None

Return Value

None

rsvp_api_trunk_get_by_name

This call implements P2MP trunk creation. Namespace is the same for both P2P and P2MP Trunks. Use this API to check whether any tunnel already exists with same name.

Syntax

```
struct rsvp_trunk * rsvp_api_trunk_get_by_name (char *name, u_char family,
                                                u_char trunk_type, int *status);
```

Input Parameters

`name`	Unique name with which to create the trunk
`family`	AF_INET
`trunk_type`	Set to RSVP_TRUNK_TYPE_MPLS_P2MP to create P2MP trunk

Output Parameters

`status`	SUCCESS or FAILURE

Return Value

`struct rsvp_trunk` Created RSVP P2MP Trunk

rsvp_api_trunk_ingress_p2mp_session_get

This call helps to create or fetch a P2MP LSP.

Syntax

```
struct rsvp_p2mp_session *
rsvp_api_trunk_ingress_p2mp_session_get (struct rsvp_trunk *trunk,
```

```
                                        u_char session_type,
                                        bool_t _create)
```

Input Parameters

`trunk`	RSVP P2MP trunk
`session_type`	Type of session; one of: RSVP_SESSION_TYPE_PRIMARY RSVP_SESSION_TYPE_SECONDARY
`t _create`	Set to RSVP_TRUE to create newly.

Output Parameters

None

Return Value

`struct rsvp_p2mp_session` P2MP LSP session

rsvp_api_trunk_p2mp_set_ext_tnl_id

This call sets an extended tunnel address for a P2MP trunk.

Syntax

```
Int rsvp_api_trunk_p2mp_set_ ext_tnl_id (struct rsvp_trunk *trunk, struct prefix *p)
```

Input Parameters

`trunk`	RSVP P2MP trunk
`p`	Prefix

Output Parameters

None

Return Value

RSVP_ERR_INVALID_ARGS when arguments are invalid

RSVP_SUCCESS when the call is successful

rsvp_api_trunk_p2mp_set_filter

This call sets the reservation style for a P2MP trunk.

Syntax

```
void
rsvp_api_trunk_p2mp_set_filter (struct rsvp_trunk *trunk, u_char style)
```

Input Parameters

`trunk`	RSVP P2MP trunk
`style`	Reservation style, either FF or SE

Output Parameters

None

Return Value

None

rsvp_api_trunk_p2mp_set_ingress

This call sets an ingress address for a P2MP trunk.

Syntax

```
Int rsvp_api_trunk_p2mp_set_ingress (struct rsvp_trunk *trunk, struct prefix *p)
```

Input Parameters

`trunk`	RSVP P2MP trunk
`p`	Prefix

Output Parameters

None

Return Value

RSVP_ERR_INVALID_ARGS when arguments are invalid

RSVP_SUCCESS when the call is successful

rsvp_api_trunk_restart_all_sessions

This call clears the RSVP P2MP trunk and related LSPs.

Syntax

```
void
rsvp_api_trunk_restart_all_sessions (struct rsvp_trunk *trunk,
                                     session_role_t session_role,
                                     u_char session_type)
```

Input Parameters

`trunk`	RSVP P2MP trunk to clear
`session_role`	Session role
`session_type`	PRIMARY or SECONDARY

Note: Specify session role as RSVP_SESSION_ROLE_NONE and `session_type` as RSVP_SESSION_TYPE_UNDEFINED to restart trunk. Specify session type as PRIMARY or SECONDARY to clear a specific P2MP LSP.

Output Parameters

None

Return Value

None

rsvp_api_trunk_s2l_session_get

This call creates a P2MP S2L session.

Syntax

```
struct rsvp_session *
rsvp_api_trunk_s2l_session_get (struct rsvp_p2mp_session *p2mp_session,
                                struct prefix *egress, bool_t _create)
```

Input Parameters

`session`	P2MP session
`egress`	S2L egress
`t _create`	Set to RSVP_TRUE to create new session

Note: Use the same API with create as RSVP_FALSE for P2MP S2L LSP lookup.

Output Parameters

None

Return Value

`struct rsvp_session` P2MP S2L session

rsvp_p2mp_session_class_type_add

This call sets a DiffServ TE (DSTE) class type for a P2MP session.

Syntax

```
void
rsvp_p2mp_session_class_type_add (struct rsvp_p2mp_session *orig_session,
                                  char *name)
```

Input Parameters

`orig_session`	P2MP session
`name`	DSTE class name

Output Parameters

None

Return Value

None

rsvp_p2mp_session_class_type_del

This call removes a configured class type for a P2MP session.

Syntax

```
void
rsvp_p2mp_session_class_type_del  (struct rsvp_p2mp_session *orig_session)
```

Input Parameters

`orig_session`	P2MP session

Output Parameters

None

Return Value

None

rsvp_s2l_session_ingress_restart

This call clears an RSVP P2MPS2L LSP.

Syntax

```
void
rsvp_s2l_session_ingress_restart (struct rsvp_session *session)
```

Input Parameters

`session`	P2MP S2L session

Output Parameters

None

Return Value

None

Index

N

O

P

Q

R

S

T