MPLS Forwarding Service APIs with Diffserv and TE Extensions Implementation Agreement

Revision 1.0

Editors:

Manikantan Srinivasan, manis@futsoft.com
Reda Haddad, reda.haddad@ericsson.com

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the remainder of this document are to be interpreted as described in the NPF Software API Conventions Implementation Agreement revision 2.0.

For additional information contact:
The Network Processing Forum, 39355 California Street,
Suite 307, Fremont, CA 94538
+1 510 608-5990 phone info@npforum.org

Table of Contents

Table of Figures

1 Revision History

Revision	Date	Reason for Changes
1.0	09/23/2003	Created Rev 1.0 of the implementation agreement by taking the MPLS Forwarding Service APIs with Diffserv and TE Extensions (npf2003.272.05) and making minor editorial corrections.

2 Introduction

In MPLS networks, data transmission occurs on label-switched paths (LSPs). An LSP is a path through a sequence of interconnected routers, which forward packets from the start to the end of the path based on labels. LSPs are established prior to data transmission – driven by the control plane - or upon detection of a certain flow of data – driven by the data plane. The labels, which are underlying protocol-specific identifiers, are distributed using Label Distribution Protocol (LDP) or Resource Reservation Protocol (RSVP) or piggybacked on routing protocols like Border Gateway Protocol (BGP).

Data forwarding or handling based on label information is carried out at the following types of devices:

- MPLS Ingress Label Edge Router (LER)
- MPLS Intermediate or Core Label Switching Router (LSR)
- MPLS Egress Label Edge Router (LER)

In the ingress LER, the MPLS forwarding engine/module classifies the data being forwarded into Forwarding Equivalence Classes (FECs). Once the FEC to which a packet belongs is determined, the LER uses the Next Hop Label Forwarding Entry (NHLFE) associated with that FEC to forward the packet onto an LSP. The intermediate LSR and the egress LER use the Incoming Label Map (ILM) and the associated Next Hop Label Forwarding Entry (NHLFE) for handling and forwarding the labeled packet.

Figure 1 depicts the typical architecture / relationship between MPLS signaling and NPF APIs.

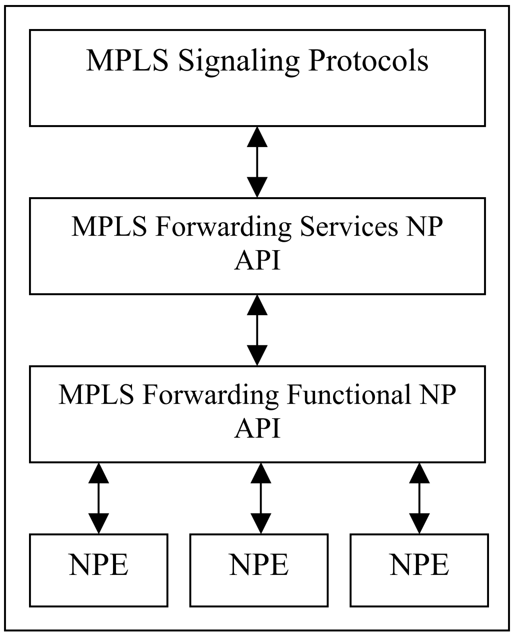
MPLS Signaling Protocols
MPLS Forwarding Services NP API
MPLS Forwarding Functional NP API
NPE
NPE
NPE

Figure 1 - MPLS Signaling Protocols - NP API architectural relationship

The MPLS Forwarding Services API provides a generic interface for configuring and managing the forwarding plane of the MPLS layer. The signaling protocols like LDP, RSVP-TE etc., use the function calls of this API to configure and manage MPLS forwarding information.

2.1 Scope

The MPLS Forwarding Services API provides function calls for the control and management of data transmission over MPLS label switched paths in the data/forwarding plane. The scope of this API defined is restricted to LSPs associated with Ethernet (Generic), ATM, FR and POS (generic) interfaces / networks.

2.2 Abbreviations and Acronyms

ATM	Asynchronous Transfer Mode
DSCP	Diffserv Code Point
FEC	Forwarding Equivalence Class
FR	Frame Relay
FTN	FEC to NHLFE
ILM	Incoming Label Map
LSP	Label Switched Path
MPLS	Multi Protocol Label Switching
NHLFE	Next Hop Label Forwarding Entry
PHB	Per Hop Behavior
PSC	PHB Scheduling Class
SAPI	Services API
TE	Traffic Engineering

3 Assumptions

The following assumptions have been made in designing the MPLS forwarding APIs and the associated data structures

1. Some MPLS forwarding designs might maintain an FTN and ILM database per interface to enable faster and easy lookups.
2. An FTN entry is uniquely identified based on the FEC parameters. In basic MPLS forwarding, the FEC parameter is either an IPv4/IPv6 prefix or an IPv4/IPv6 host address.
3. An FTN entry information might be stored as part of IP forwarding database, where the next hop information will be an NHLFE Set.
4. An ILM entry is uniquely identified based on incoming label and incoming interface identifier pair. In cases where the label is part of per-platform label space, the incoming interface identifier will have a value 0.
5. An NHLFE entry is uniquely identified based on outgoing interface identifier, a next hop IP address and an outgoing label Stack. The outgoing label will be the top label in the label stack information associated with the NHLFE.
6. For an LSP there can be multiple NHLFEs. A policy may be defined to select between the NHLFEs. The policy information and the associated NHLFEs are contained within an NHLFE Set.
7. As table or database information, the basic forwarding design assumes an LSP table/database, an NHLFE table/database, and an NHLFE Set table/database.
8. An NHLFE/NHLFE Set can be created first in the NHLFE/NHLFE Set table/database. The creation will result in a unique NHLFE/NHLFE Set handle. The NHLFE handle value will be passed along with the LSP creation as part of the NHLFE Set information
9. An NHLFE/NHLFE Set can also be created along with a LSP, when the LSP is created. An NHLFE Set will be created even if it is just one NHLFE.
10. An NHLFE/NHLFE Set can be shared by multiple LSPs, if the system can support label merging. We assume a label space may be partitioned into multiple non-overlapping signaling protocol partitions.
11. An NHLFE Set can be modified with a updated set of NHLFE information. When an NHLFE Set is modified, this modification is reflected on all the LSPs that are associated with the NHLFE Set that is modified.
12. At an ingress of a LSP, (where the data forwarding is based on the FEC), it is possible to have multiple NHLFEs. Forwarding data over different NHLFEs for providing priority handling of data packets or for load balancing can be carried out at the LSP ingress. This will be based on suitable policies (examples given below). Forwarding information will be maintained as an array of policy information, consisting of NHLFE handles and weights associated with those Next Hops, as shown in the weight policy example below. The set of NHLFEs, array of policy information are part of an NHLFE Set.

Weight	1	3	10	16
NHLFE Handle	1	2	3	4

13. When using less than the number of bits in a variable (for example, 20 bits for generic or shim label) we assume the least significant bits are used and the remainder of the bits are padded with 0s, unless otherwise specified.
14. `NPF_errorReporting_t mplsErrorReporting` – An input parameter that is passed in most of the MPLS SAPI API function call, indicates the degree of error reporting desired from the callbacks performed at the completion of that function. For more information, please refer section 5.2 in NPF Software API Conventions Implementation Agreement – Revision 1.0

Diffserv and TE Assumptions

1. Compliance to RFC 2474, RFC 2475, RFC 3140, and RFC 3270.
2. The means used to forward packets from ingress to egress interfaces, or "fabric", may have the capability of differentiating services otherwise referred to as Fabric Classes of Services (FCoS).
3. If FCoS is supported, the FCoS to use for a particular packet is derived from and only from a DSCP value; The MPLS label + EXP-bits inferring a PHB, have to be transformed into a DSCP equivalent in order to cross the fabric on a particular FCoS. The setting of the platform wide "DSCP to FCoS" map is beyond the scope of this document and assumed to be under the Differentiated Services Control or SAPI.
4. Classification is not MPLS functionality and classifiers need to be configured by other SAPIs. More specifically, extended FECs are not considered as part of the MPLS SAPI. For example, Diffserv may configure a multi-field classifier to forward packets into an MPLS tunnel/LSP (PBR [reference NPF 2003.139]).
5. On an ingress LER, the DSCP extracted from a packet may be interpreted before being presented to the MPLS subsystem (possibly remarked). The interpreted DSCP will be "trusted" and used as needed by the MPLS subsystem. The interpretation of the DSCP is not the responsibility of MPLS, rather the responsibility of Diffserv.
6. The interpreted DSCP, or the MPLS inferred DSCP is carried to the egress side as part of the packet metadata. The MPLS inferred DSCP can be extracted from either the top or the inner label (or the IPv4 header) depending on the Diffserv LSP model.
7. A "DSCP to EXP" table is available per NHLFE. Multiple NHLFE entries can use the same "DSCP to EXP" table.
8. An "EXP to DSCP" table is available per ILM entry. Multiple ILM entries can use the same "EXP to DSCP" table.
9. At Egress LER, the choice of using the "LSP Diffserv information" from the top label or the "tunneled Diffserv information" from the encapsulated packet can be configured. The Extracted Diffserv information, either the LSP or tunneled, can then be used to determine the PHB. The forwarded Diffserv information can either be the LSP or the tunneled information, and may be different than the one used to determine the PHB.
10. At Egress interface, the choice of keeping the EXP from the encapsulated packet or overriding with EXP in the NHLFE entry or overriding with EXP bits extracted from a DSCP to EXP table, can be configured.

4 External Requirements/Dependencies

The current document scope does not cover the following

1. Dependency, if any, for an LSP creation of type FTN entry being carried out with the help of IP forwarding APIs.
2. Enabling an interface as MPLS data handling (receiving labeled packets and forwarding labeled packets) interface. It is assumed this is done as part of the Interfaces Management API [4].
3. Enabling an MPLS Tunnel as a valid interface and making it visible to routing protocols for their advertisements. It is assumed this is done as part of the Interfaces Management API [4].

5 Data Types

This section describes the MPLS SAPI data structure definitions. Figure 2 below gives the relationships between the data structures defined in this document.

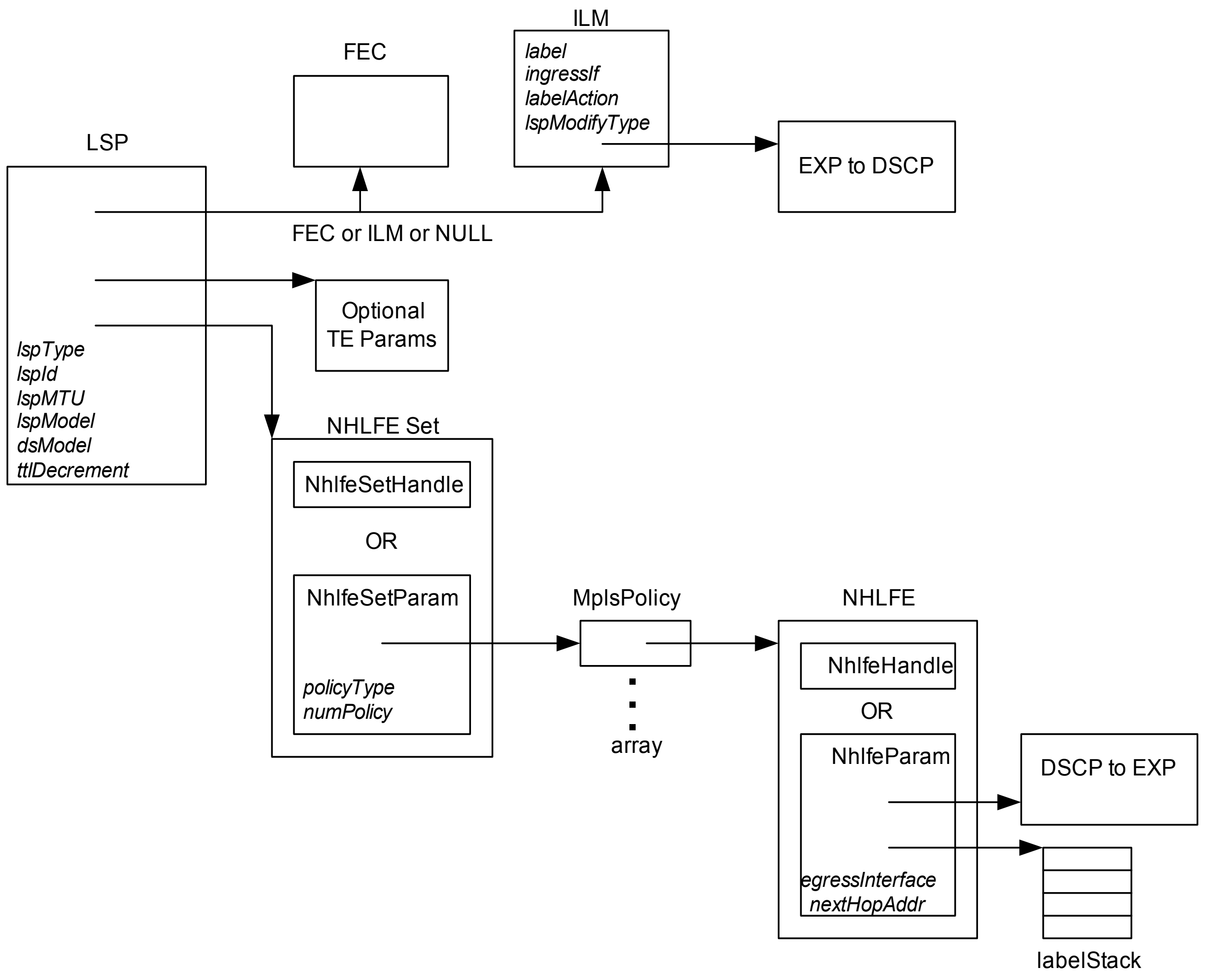

Figure 2 - Relationship / Dependency of MPLS NP API - Data Structures

5.1 NPF MPLS Common Data Types

5.1.1 NPF_MPLS_IP_Type_t

This enumerated type definition is used to indicate whether the IP protocol is IPv4 or IPv6.

```
typedef enum {
   NPF_MPLS_IPV4 = 1,
   NPF_MPLS_IPV6 = 2
} NPF_MPLS_IP_Type_t;
```

5.1.2 NPF_MPLS_HostAddr_t

This structure type definition is used to store either an IPv4 Host address or an IPv6 Host address.

```
typedef struct{
   NPF_MPLS_IP_Type_t   ipAddrType;
   union {
      NPF_IPv4Address_t ipv4DestHostAddr; /* IPv4 Host Address */
      NPF_IPv6Address_t ipv6DestHostAddr; /* IPv6 Host Address */
   } u;
} NPF_MPLS_HostAddr_t;
```

5.2 NPF MPLS Label Data Types

5.2.1 NPF_MPLS_LabelType_t

This enumerated type definition is used to indicate the label type as shim label (generic label), or an ATM label or an FR label.

```
typedef enum {
   NPF_MPLS_LABEL_TYPE_GENERIC = 1,
   NPF_MPLS_LABEL_TYPE_ATM     = 2,
   NPF_MPLS_LABEL_TYPE_FR      = 3
} NPF_MPLS_LabelType_t;
```

5.2.2 NPF_MPLS_ShimLabel_t

This structure type definition is used to store a MPLS Generic label value. A MPLS label, as used in this specification, refers to a 32-bit Label Stack Entry encoding as specified in RFC 3032.

```
typedef NPF_uint32_t NPF_MPLS_ShimLabel_t;
```

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Value																				**EXP**			**S**	**TTL**							

Subfield	Usage	Size (bits)
Value	Label Value	20
EXP	Experimental Use	3
S	Bottom of Stack Indicator	1
TTL	Time to Live	8

Figure 3 - Label Encoding

Note: This structure is specified as is to identify the position of the label bits within the 32-bit field. TTL, S and EXP bits are not required to be defined as part of the shim label value. Implementer can make use of the TTL, S and EXP bits in their implementation.

5.2.3 NPF_MPLS_ATM_Label_t

This structure type definition is used to store a MPLS ATM label value. This definition is based on the ATM VC definition made in NP Forum – Interface Management API Implementation Agreement Revision 1.0.

```
typedef NPF_VccAddr_t NPF_MPLS_ATM_Label_t
```

5.2.4 NPF_MPLS_DLCI_Type_t

This enumerated type definition is used to identify MPLS FR label value as 10 bits DLCI or 23-bit DLCI.

```
typedef enum {
   NPF_MPLS_DLCI_10 = 1,
   NPF_MPLS_DLCI_23 = 2
} NPF_MPLS_DLCI_Type_t;
```

5.2.5 NPF_MPLS_FR_Label_t

This structure type definition is used to store an MPLS FR label value.

```
typedef struct {
   NPF_MPLS_DLCI_Type dlciType; /* Length of the DLCI in bits */
   NPF_uint32_t dlci;           /* DLCI */
} NPF_MPLS_FR_Label_t;
```

5.2.6 NPF_MPLS_Label_t

This structure type definition is used to store an MPLS label value.

```
typedef struct {
   NPF_MPLS_LabelType_t labelType;    /* Type of label      */
   union {
      NPF_MPLS_ShimLabel_t shimLabel; /* Generic label      */
      NPF_MPLS_ATM_Label_t atmLabel;  /* ATM label          */
      NPF_MPLS_FR_Label_t  frLabel;   /* Frame Relay Label */
   } u;
} NPF_MPLS_Label_t;
```

5.2.7 NPF_MPLS_LabelStack_t

This structure type definition is used to store MPLS label stack information.

```
typedef struct {
    NPF_int32_t       numLabels;  /* Number of labels */
    NPF_MPLS_Label_t* labelStack; /* Stack of labels  */
} NPF_MPLS_LabelStack_t;
```

5.3 NPF MPLS Diffserv Data Types

5.3.1 NPF_MPLS_DSCP_EXP_Entry_t

This structure type definition relates a DSCP value to an EXP value that can be used as a map entry for a DSCP to an EXP or vice versa.

```
typedef struct {
    NPF_uint8_t dscp; /*DSCP value */
    NPF_uint8_t exp;  /*EXP value  */
} NPF_MPLS_DSCP_EXP_Entry_t;
```

5.3.2 NPF_MPLS_DSCP_EXP_Param_t

This structure type definition contains information associated with a DSCP EXP map.

```
typedef struct {
    NPF_uint32_t tableId;    /* unique id set by user */
    NPF_uint8_t  numEntries; /* Number of entries in this map */
    NPF_MPLS_DSCP_EXP_Entry_t *entries; /* DSCP to EXP entries array*/
} NPF_MPLS_DSCP_EXP_Param_t;
```

5.3.3 NPF_MPLS_DSCP_EXP_Handle_t

This structure type definition holds DSCP EXP map handle information

```
typedef NPF_uint32_t NPF_MPLS_DSCP_EXP_Handle_t;
```

5.3.4 NPF_MPLS_DSCP_EXP_InfoType_t

This enumerated type definition is used to indicate whether a DSCP EXP map handle is provided or set of parameters that is associated with a DSCP EXP map is provided during the DSCP EXP map creation.

```
typedef enum {
   NPF_MPLS_DSCPEXP_HANDLE = 1,
   NPF_MPLS_DSCPEXP_PARAMS = 2
} NPF_MPLS_DSCP_EXP_InfoType_t;
```

5.3.5 NPF_MPLS_DSCP_EXP_Type_t

This enumerated type definition is used to indicate whether a DSCP EXP map is a DSCP to EXP or an EXP to DSCP type.

```
typedef enum {
   NPF_MPLS_DSCPEXP_DTOE = 1,
   NPF_MPLS_DSCPEXP_ETOD = 2
} NPF_MPLS_DSCP_EXP_Type_t;
```

5.3.6 NPF_MPLS_DSCP_EXP_t

This structure type definition is used to provide the information stored as part of a DSCP to EXP or an EXP to DSCP map.

```
typedef struct {
   NPF_MPLS_DSCP_EXP_Type_t      type;      /* DSCP to Exp or Exp to DSCP*/
   NPF_MPLS_DSCP_EXP_InfoType_t  paramType; /* Handle or map information */
   union {
      NPF_MPLS_DSCP_EXP_Handle_t  mapHandle; /* Map Handle */
      NPF_MPLS_DSCP_EXP_Param_t   mapParam;  /* Map Parameters */
   } u;
} NPF_MPLS_DSCP_EXP_t;
```

5.3.7 NPF_MPLS_PSC_ID_t

This structure type definition holds the PSC ID as defined in RFC3140 section 2.

```
typedef NPF_uint16_t NPF_MPLS_PSC_ID_t;
```

5.3.8 NPF_MPLS_PSC_ID_Value_t

This enumerated type definition is used to provide the PSC ID values. The only predefined PSC ID is for Best Effort service.

```
typedef enum {
   NPF_MPLS_PSCID_BE = 0
} NPF_MPLS_PSC_ID_Value_t;
```

5.3.9 NPF_MPLS_DS_LSP_Type_t

This enumerated type definition is used to provide the LSP type as either ELSP or LLSP.

```
typedef enum {
   NPF_MPLS_DS_LSP_TYPE_NONE = 0, /* non-Diffserv type */
   NPF_MPLS_DS_LSP_TYPE_ELSP = 1, /* ELSP */
   NPF_MPLS_DS_LSP_TYPE_LLSP = 2  /* LLSP */
} NPF_MPLS_DS_LSP_Type_t;
```

5.3.10 NPF_MPLS_DS_LSP_Model_t

This enumerated type definition is used to provide the LSP model as either pipe, short pipe or uniform as defined in RFC 3270.

```
typedef enum {
   NPF_MPLS_DS_LSP_MODEL_NONE      = 0, /* non-Diffserv model */
   NPF_MPLS_DS_LSP_MODEL_PIPE      = 1, /* Pipe */
   NPF_MPLS_DS_LSP_MODEL_SHORTPIPE = 2, /* Short-pipe */
   NPF_MPLS_DS_LSP_MODEL_UNIFORM   = 3  /* Uniform */
} NPF_MPLS_DS_LSP_Model_t;
```

5.4 NPF MPLS TE Data Types

5.4.1 NPF MPLS_TE_Param_t

This structure type definition is used to provide the parameters stored as part of a TE tunnel.

```
typedef struct {
   NPF_uint32_t maxRate;       /*Max or Peak rate (bps)*/
   NPF_uint32_t meanRate;      /*Mean rate (bps)*/
   NPF_uint32_t maxBurstSize;  /*Max Burst size in bytes*/
   NPF_uint32_t meanBurstSize; /*Mean Burst size in bytes*/
   NPF_uint32_t exBurstSize;   /*Excess Burst size in bytes*/
   NPF_uint32_t frequency;     /*Frequency of token refresh*/
   NPF_uint8_t  weight;        /*Weight associated with tunnel*/
   NPF_uint8_t  trafficClass;  /*Derived from parameters above*/
} NPF_MPLS_TE_Param_t;
```

Note: the traffic class table can be found in Appendix B.

5.5 NPF MPLS NHLFE Data Types

5.5.1 NPF_MPLS_NHLFE_Handle_t

This structure type definition holds NHLFE handle information.

```
typedef NPF_uint32_t NPF_MPLS_NHLFE_Handle_t;
```

5.5.2 NPF_MPLS_NHLFE_Param_t

This structure type definition contains information associated with an NHLFE.

```
typedef struct {
   NPF_IfHandle_t        egressInterface; /* Outgoing interface */
   NPF_MPLS_HostAddr_t   nextHopAddr;     /* Next Hop IPv4/IPv6 address */
   NPF_MPLS_LabelStack_t labelStack;      /* label stack to be pushed*/
   NPF_MPLS_DSCP_EXP_t   *dscpToExp;      /* DSCP to EXP map */
} NPF_MPLS_NHLFE_Param_t;
```

Note: If the value of the dscpToExp pointer is NULL then the NHLFE is non-DS enabled. If the pointer is not NULL then the NHLFE is DS enabled.

5.5.3 NPF_MPLS_NHLFE_InfoType_t Type

This enumerated type definition is used to indicate whether an NHLFE handle is provided or set of parameters that is associated with a NHLFE is provided during the NHLFE creation.

```
typedef enum {
   NPF_MPLS_NHLFE_HANDLE = 1,
   NPF_MPLS_NHLFE_PARAMS = 2
} NPF_MPLS_NHLFE_InfoType_t;
```

5.5.4 NPF_MPLS_NHLFE_t

This structure type definition is used to provide the information stored as part of an NHLFE.

```
typedef struct {
   NPF_MPLS_NHLFE_InfoType_t  paramType;   /* Handle or NHLFE information */
   union {
      NPF_MPLS_NHLFE_Handle_t nhlfeHandle; /* NHLFE Handle */
      NPF_MPLS_NHLFE_Param_t  nhlfeParam;  /* NHLFE Parameters */
    } u;
} NPF_MPLS_NHLFE_t;
```

5.5.5 NPF_MPLS_NHLFE_SET_PolicyType_t

This enumerated type definition specifies the NHLFE policy type associated with each NHLFE Set.

```
typedef enum {
   NPF_MPLS_POLICYTYPE_NONE   = 0,
   NPF_MPLS_POLICYTYPE_WEIGHT = 1,
   NPF_MPLS_POLICYTYPE_ELSP   = 2,
   NPF_MPLS_POLICYTYPE_LLSP   = 3
} NPF_MPLS_NHLFE_SET_PolicyType_t;
```

5.5.6 NPF_MPLS_WeightPolicy_t

This structure type definition specifies a weight policy table entry corresponding to a policy type.

```
typedef struct{
   NPF_uint32_t weight;
} NPF_MPLS_WeightPolicy_t;
```

5.5.7 NPF_MPLS_DS_Policy_t

This structure type definition specifies a Diffserv policy table entry corresponding to a policy type.

```
typedef struct{
   NPF_uint8_t       dscp; /* incoming DSCP to select on */
} NPF_MPLS_DS_Policy_t;
```

Note: Use the 6 MSB for the `dscp`. The LSB conveys whether it is a PHB (0) or a PSC (1). The dscp here can also infer a PSC as specified in RFC3140, where we may use the 3 MSB bits of the `dscp` to convey the PSC.

5.5.8 NPF_MPLS_Policy_t

This structure type definition specifies a policy entry.

```
typedef struct{
   NPF_MPLS_NHLFE_t *nhlfe;
   union {
      NPF_MPLS_WeightPolicy_t weightPolicy;
      NPF_MPLS_DS_Policy_t    dsPolicy;
   } u;
} NPF_MPLS_Policy_t;
```

5.5.9 NPF_MPLS_NHLFE_SET_Param_t

This structure specifies the parameters used in an NHLFE Set to determine which NHLFE(s) are used. The parameters consist of a number of policies of a particular policy type. For example, for the load balancing case, the policy type would be set to NPF_MPLS_POLICYTYPE_WEIGHT. The union information inside the policy structure (NPF_MPLS_Policy_t) will be interpreted as weight policy (NPF_MPLS_WeightPolicy_t). The weight policy field (NPF_MPLS_WeightPolicy_t) carry the weights associated with each NHLFE, see section 5.5.6.

In the case of an EXP to NHLFE mapping, the policy array would have eight entries, indexed by the shim exp bits and specifying the associated NHLFE (2 different exp values can map to the same NHLFE).

```
typedef struct {
   NPF_uint32_t                   setId;
   NPF_MPLS_NHLFE_SET_PolicyType_t policyType;
   NPF_uint32_t                   numPolicy;
   NPF_MPLS_Policy_t              **policyArray;
} NPF_MPLS_NHLFE_SET_Param_t;
```

Note: the SetId uniquely identifies a NHLFE set per user context.

5.5.10 NPF_MPLS_NHLFE_SET_Handle_t

This structure type definition holds NHLFE Set handle information.

```
typedef NPF_uint32_t NPF_MPLS_NHLFE_SET_Handle_t;
```

5.5.11 NPF_MPLS_NHLFE_SET_Type_t

This enumerated type definition is used to indicate whether NHLFE Set handle or NHLFE Set information are selected.

```
typedef enum {
   NPF_MPLS_NHLFESET_HANDLE = 1,
   NPF_MPLS_NHLFESET_PARAMS = 2
} NPF_MPLS_NHLFE_SET_Type_t;
```

5.5.12 NPF_MPLS_NHLFE_SET_t

This structure type definition specifies the information stored as part of an NHLFE Set.

```
typedef struct {
   NPF_MPLS_NHLFE_SET_Type_t setType;  /* Handle or NHLFE Set information */
   union {
      NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;/* NHLFE Set Handle      */
      NPF_MPLS_NHLFE_SET_Param_t  nhlfeSetParam; /* NHLFE Set Parameters  */
    } u;
} NPF_MPLS_NHLFE_SET_t;
```

5.6 NPF MPLS FEC Data Types

5.6.1 NPF_MPLS_FEC_Param_t

This structure type definition is used to store MPLS FEC parameter information. It stores one of the four FEC parameters – IPv4 prefix, IPv6 prefix, IPv4host address or IPv6 host address.

```
typedef struct{
   union {
      NPF_IPv4Prefix_t ipv4DestNetPrefix;   /* IPv4 prefix       */
      NPF_IPv6Prefix_t ipv6DestNetPrefix;   /* IPv6 prefix       */
      NPF_IPv4Address_t ipv4DestHostAddr;   /* IPv4 Host Address */
      NPF_IPv6Address_t ipv6DestHostAddr;   /* IPv6 Host Address */
   } u;
} NPF_MPLS_FEC_Param_t;
```

Note: `NPF_IPv4Prefix_t` and `NPF_IPv6Prefix_t` are (to be) defined in the NPF Conventions IA. Please refer to the informative Appendix C for the relationship of MPLS SAPI with IPv4/IPv6 SAPI for FTN mapping.

5.6.2 NPF_MPLS_FEC_Type_t

This enumerated type definition specifies the FEC classification information.

```
typedef enum {
   NPF_MPLS_FEC_IPV4_DEST_PREFIX = 1, /* IPv4 prefix       */
   NPF_MPLS_FEC_IPV4_HOSTADDR    = 2, /* IPv4 Host Address */
   NPF_MPLS_FEC_IPV6_DEST_PREFIX = 3, /* IPv6 prefix       */
   NPF_MPLS_FEC_IPV6_HOSTADDR    = 4  /* IPv6 Host Address */
} NPF_MPLS_FEC_Type_t;
```

5.6.3 NPF_MPLS_FEC_t

This structure type definition contains the FEC information.

```
typedef struct{
   NPF_MPLS_FEC_Type_t  fecType;
   NPF_MPLS_FEC_Param_t param;
} NPF_MPLS_FEC_t;
```

Note: Source Port and Destination port numbers have been added as elements in mplsFTN entry defined in the MPLS FTN MIB. However, the basic MPLS signaling protocol – LDP does not have support for TLVs to convey the Port information. Hence, the FEC definition here does not include Port numbers.

5.7 NPF MPLS Label Action Data Types

5.7.1 NPF_MPLS_Modifier_t

This enumerated type definition specifies the additional processing to be done on a data packet as part of the FTN or ILM handling.

```
typedef enum {
   NPF_MPLS_REDIRECT             = 1,
   NPF_MPLS_COPY_PROCESS_OPCODE = 2
} NPF_MPLS_Modifier_t;
```

`NPF_MPLS_REDIRECT` : This indicates that a the data packet be redirected for further processing. Example usage - Used for testing such as continuity checking of LSPs or sending test packets on LSPs that should not be forwarded out to an LSP user.
`NPF_MPLS_COPY_PROCESS_OPCODE` : This indicates that data packets are to be duplicated and redirected for further processing. Example usage - Used for LSP Ping and trace route, lawful intercept, monitoring and debugging.

5.7.2 NPF_MPLS_LabelAction_t

This enumerated type definition specifies the FEC classification information.

```
typedef enum {
   NPF_MPLS_POP_AND_LOOKUP     = 1,
   NPF_MPLS_POP_AND_FORWARD    = 2,
   NPF_MPLS_NO_POP_AND_FORWARD = 3,
   NPF_MPLS_DISCARD            = 4
} NPF_MPLS_LabelAction_t;
```

`NPF_MPLS_POP_AND_LOOKUP`: This indicates that the top label needs to be popped and lookup should be done on the next header (either another label or an IPv4 header).

`NPF_MPLS_POP_AND_FORWARD`: This indicates that the top label should be swapped or replaced with the new label and the packet forwarded.

`NPF_MPLS_NO_POP_AND_FORWARD`: This indicates that the new label should be pushed onto the packet.

`NPF_MPLS_DISCARD`: This indicates that the packet should be discarded/dropped.

NOTE: To support Penultimate hop popping, the Penultimate node pops the incoming label and forwards the packet based on the information in the popped label. The ILM entry with opcode `NPF_MPLS_POP_AND_FORWARD` points to the NHLFE containing a label with a value of `IMPLICIT_NULL(3).` This indicates that no label is pushed at egress. If no labels remain in the packet's label stack, the packet is sent unlabeled on the egress interface with the appropriate L2 encapsulation for an IP packet, otherwise the packet is sent on the egress interface with the appropriate L2 encapsulation for an MPLS packet.

5.8 NPF MPLS ILM Data Types

5.8.1 NPF_MPLS_ILM_t

This structure type definition specifies the information stored as part of an ILM.

```
typedef struct {
   NPF_MPLS_Label_t       incomingLabel;    /* Incoming label*/
   NPF_IfHandle_t         ingressInterface; /* Incoming interface */
   NPF_MPLS_LabelAction_t labelAction;      /* Label action */
   NPF_MPLS_Modifier_t    lspModifyType;    /* Additional processing during
                                               the handling of packet */
   NPF_MPLS_DSCP_EXP_t    *expToDscp;       /* EXP to DSCP table associated
                                               with ELSP*/
} NPF_MPLS_ILM_t;
```

Note: the incomingLabel and ingressInterface are used to uniquely identify an ILM entry. Setting the `ingressInterface` to `NULL` specifies a platform wide label.

5.9 NPF MPLS LSP Data Types

5.9.1 NPF_MPLS_LSP_Type_t

This enumerated type definition specifies the LSP type.

```
typedef enum{
   NPF_MPLS_LSP_FEC = 1, /*Associates FEC with NHLFE */
   NPF_MPLS_LSP_ILM = 2  /*Associates ILM with NHLFE */
   NPF_MPLS_LSP_TUN = 3  /*Creates a tunnel endpoint */
} NPF_MPLS_LSP_Type_t;
```

5.9.2 NPF_MPLS_LSP_Id_t

This structure type definition holds the LSP ID value. The LSPID parameter information is used with CRLDP and RSVP-TE.

```
typedef NPF_uint32_t NPF_MPLS_LSP_Id_t;
```

5.9.3 NPF_MPLS_LSP_t

This structure definition specifies the information required to create LSP. The structure is used to associate incoming label with one or more NHLFE entries in an NHLFE SET or to associate an incoming FEC to one or more NHLFE entries in an NHLFE SET. The NHLFE Set can be NULL in case of Egress LER.

```
typedef struct {
   NPF_MPLS_LSP_Type_t     lspType;     /* Type of LSP*/
   NPF_MPLS_LSP_Id_t       lspId;       /*LSP Tunnel Parameter - Identifier*/
   NPF_MPLS_TE_Param_t     *teParams;   /*tunnel/LSP parameters*/
   NPF_uint16_t            lspMtu;      /*LSP MTU */
   union {
      NPF_MPLS_FEC_t       *fec;        /* FEC */
      NPF_MPLS_ILM_t       *ilm;        /* ILM */
   } u;
   NPF_MPLS_DS_LSP_Model_t dsModel;      /*pipe, short pipe or uniform  */
   NPF_MPLS_DS_LSP_Type_t  dsLspType;    /*E-LSP, L-LSP, none           */
   NPF_uint16_t            ttlDecrement; /*let SAPI or below figure out
                                          where to decrement           */
   NPF_MPLS_NHLFE_SET_t    *nhlfeSet;    /* Associated NHLFE Set        */
} NPF_MPLS_LSP_t;
```

The key values used to uniquely identify an LSP are as follows:

- An FTN entry is uniquely identified based on the FEC parameters.
- An ILM entry is uniquely identified based on incoming label and incoming interface identifier pair.
- A tunnel is uniquely identified based on the LSP ID. In this case FEC and ILM will be NULL.

5.9.4 NPF_MPLS_LSP_Handle_t

This structure type definition holds the LSP Handle value

```
typedef NPF_uint32_t NPF_MPLS_LSP_Handle_t;
```

Note: The handle must be unique per platform, since other applications can access and use the LSP.

5.9.5 NPF_MPLS_LSP_InfoType_t

This enumerated type definition is used to indicate whether an LSP handle is provided or the parameters that define an LSP is provided.

```
typedef enum {
   NPF_MPLS_LSP_HANDLE = 1,
   NPF_MPLS_LSP_PARAMS = 2
} NPF_MPLS_LSP_InfoType_t;
```

5.9.6 NPF_MPLS_LSP_Info_t

This structure type definition is used to identify an LSP either through the LSP handle returned by the SAPI when the LSP was created or through the parameters used to create the LSP. In order to identify an LSP, only the key values must be provided. Either the LSP handle or the LSP key values may be used to retrieve the full `NPF_MPLS_LSP_t` from the SAPI.

```
typedef struct {
   NPF_MPLS_LSP_InfoType_t  paramType; /* Handle or LSP information */
   union {
      NPF_MPLS_LSP_Handle_t lspHandle; /* LSP Handle */
      NPF_MPLS_LSP_t        lspParam;  /* LSP key values */
   } u;
} NPF_MPLS_LSP_Info_t;
```

5.10 NPF MPLS LSP Statistics Data Types

5.10.1 NPF_MPLS_FEC_Stats_t

This structure type definition holds the statistics associated with a FEC (FTN – in segment).

```
typedef struct {
   NPF_uint64_t octetsRcvd;  /* Total Rx Octets */
   NPF_uint64_t packetsRcvd; /* Total Rx Packets */
   NPF_uint64_t errors;      /* Erroneous packets discarded */
   NPF_uint64_t drops;       /* Non erroneous packets discarded */
}NPF_MPLS_FEC_Stats_t;
```

5.10.2 NPF_MPLS_ILM_Stats_t

This structure type definition holds the statistics associated with an ILM – in segment.

```
typedef struct {
   NPF_uint64_t octetsRcvd;  /* Total Rx Octets                 */
   NPF_uint64_t packetsRcvd; /* Total Rx Packets                */
   NPF_uint64_t errors;      /* Erroneous packets discarded     */
   NPF_uint64_t drops;       /* Non erroneous packets discarded */
} NPF_MPLS_ILM_Stats_t;
```

5.10.3 NPF_MPLS_DSCP_EXP_EntryStat_t

This structure type definition is used to provide the information stored as part of a DSCP to EXP or an EXP to DSCP map statistics.

```
typedef struct {
   NPF_uint64_t bytes;   /* byte count   */
   NPF_uint64_t packets; /* packet count */
} NPF_MPLS_DSCP_EXP_EntryStat_t;
```

Note: The statistics data held in this structure are dependant on the association (ILM or NHLFE) of the NPF_MPLS_DSCP_EXP_t. If the NPF_MPLS_DSCP_EXP_t is associated with an ILM the statistics reflect receive counters; If it's associated with a NHLFE it contains transmit counters.

5.10.4 NPF_MPLS_DSCP_EXP_Stats_t

This structure type definition is used to provide the information for a set of “numEntries” of DSCP to EXP or EXP to DSCP map statistics.

```
typedef struct {
   NPF_uint8_t numEntries;
   NPF_MPLS_DSCP_EXP_EntryStat_t *stats; /*stats associated with entries*/
} NPF_MPLS_DSCP_EXP_Stats_t;
```

5.10.5 NPF_MPLS_NHLFE_Stats_t

This structure type definition holds the statistics associated with an NHLFE.

```
typedef struct {
   NPF_MPLS_NHLFE_Handle_t nhlfeHandle; /* NHLFE Handle                    */
   NPF_uint64_t            octetsTxed;  /* Total Tx Octets                 */
   NPF_uint64_t            packetsTxed; /* Total Tx Packets                */
   NPF_uint64_t             errors;      /* Erroneous packets discarded     */
   NPF_uint64_t             drops;       /* Non erroneous packets discarded */
} NPF_MPLS_NHLFE_Stats_t;
```

5.10.6 NPF_MPLS_NHLFE_StatsArray_t

This structure type definition holds an array of NHLFE Statistics.

```
typedef struct {
   NPF_uint32_t           nhlfeCount;
   NPF_MPLS_NHLFE_Stats_t *nhlfeStatsArray;
}NPF_MPLS_NHLFE_StatsArray_t;
```

5.10.7 NPF_MPLS_NHLFE_SET_Stats_t

This structure type definition holds the statistics associated with an NHLFE SET.

```
typedef struct {
    NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;   /* NHLFE Set Handle */
    NPF_MPLS_NHLFE_StatsArray_t *nhlfeStatsArray  /* Array of NHLFE
                                                      Statistics    */
} NPF_MPLS_NHLFE_SET_Stats_t;
```

5.10.8 NPF_MPLS_NHLFE_SET StatsArray_t

This structure type definition holds an array of NHLFE SET Statistics.

```
typedef struct {
   NPF_uint32_t               nhlfeSetCount;
   NPF_MPLS_NHLFE_SET_Stats_t *nhlfeSetStatsArray;
} NPF_MPLS_NHLFE_SET_StatsArray_t;
```

5.10.9 NPF_MPLS_LSP_Statisitics_t

This structure type definition holds the statistics associated with an LSP.

```
typedef struct {
   NPF_MPLS_LSP_Type_t     lspType;       /* Type of LSP    */
   NPF_MPLS_LSP_Handle_t   lspHandle;     /* LSP identifier */
   union {
      NPF_MPLS_FEC_Stats_t fecStatistics; /* FEC */
      NPF_MPLS_ILM_Stats_t ilmStatistics; /* ILM */
   } u;
   NPF_MPLS_NHLFE_StatsArray_t nhlfeStatsArray; /* Associated NHLFEs*/
} NPF_MPLS_LSP_Stats_t;
```

5.11 Data Structures for Completion Callbacks

This section describes the completion callback functions and the associated data structures.

5.11.1 NPF_MPLS_CallbackType_t

This enumerated type definition specifies the call back types.

```
typedef enum NPF_MPLSCallbackType {
   NPF_MPLS_LSP_ENTRY_CREATE          = 1,
   NPF_MPLS_LSP_ENTRY_DELETE          = 2,
   NPF_MPLS_LSP_ENTRY_MODIFY          = 3,
   NPF_MPLS_LSP_ATTRIBUTE_QUERY       = 4,
   NPF_MPLS_LSP_ENTRY_QUERY           = 5,
   NPF_MPLS_LSP_STATS_QUERY           = 6,
   NPF_MPLS_NHLFE_ENTRY_CREATE        = 7,
   NPF_MPLS_NHLFE_ENTRY_DELETE        = 8,
   NPF_MPLS_NHLFE_ENTRY_MODIFY        = 9,
   NPF_MPLS_NHLFE_ATTRIBUTE_QUERY     = 10,
   NPF_MPLS_NHLFE_ENTRY_QUERY         = 11,
   NPF_MPLS_NHLFE_STATS_QUERY         = 12,
   NPF_MPLS_NHLFE_SET_CREATE          = 13,
   NPF_MPLS_NHLFE_SET_DELETE          = 14,
   NPF_MPLS_NHLFE_SET_MODIFY          = 15,
   NPF_MPLS_NHLFE_SET_ATTRIBUTE_QUERY = 16,
   NPF_MPLS_NHLFE_SET_ENTRY_QUERY     = 17,
   NPF_MPLS_NHLFE_SET_STATS_QUERY     = 18,
   NPF_MPLS_DSCPEXP_ENTRY_CREATE      = 19,
   NPF_MPLS_DSCPEXP_ENTRY_DELETE      = 20,
   NPF_MPLS_DSCPEXP_ENTRY_MODIFY      = 21,
   NPF_MPLS_DSCPEXP_ATTRIBUTE_QUERY   = 22,
   NPF_MPLS_DSCPEXP_ENTRY_QUERY       = 23,
   NPF_MPLS_DSCPEXP_STATS_QUERY       = 24
} NPF_MPLS_CallbackType_t;
```

5.11.2 NPF_MPLS_LSP_CreateResp_t

This structure type definition holds response/return information provided by the API implementation during an LSP creation callback.

```
typedef struct {
   NPF_uint32_t                    lspArrayIndex;
   NPF_MPLS_ReturnCode_t           returnCode;
   NPF_MPLS_LSP_Handle_t           lspHandle;
   NPF_MPLS_NHLFE_SET_CreateResp_t nhlfeSetResp;
   NPF_MPLS_DSCP_EXP_Handle_t      expDscpHandle;
} NPF_MPLS_LSP_CreateResp_t;
```

5.11.3 NPF_MPLS_LSP_EntryResp_t

This structure type definition holds response/return information provided by the API implementation during LSP API callbacks.

```
typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_LSP_Handle_t      lspHandle;
   NPF_MPLS_LSP_t             lspEntry;
   NPF_MPLS_DSCP_EXP_Handle_t expDscpHandle;
} NPF_MPLS_LSP_EntryResp_t;
```

5.11.4 NPF_MPLS_DSCP_EXP_CreateResp_t

This structure type definition holds response/return information provided by the API implementation during an Diffserv LSP creation callback.

```
typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_DSCP_EXP_Handle_t expDscpHandle;
} NPF_MPLS_DSCP_EXP_CreateResp_t;
```

5.11.5 NPF_MPLS_DSCP_EXP_EntryResp_t

This structure type definition holds response/return information provided by the API implementation during Diffserv LSP API callbacks.

```
typedef struct {
    NPF_uint32_t               arrayIndex;
    NPF_MPLS_ReturnCode_t      returnCode;
    NPF_MPLS_DSCP_EXP_Handle_t expDscpHandle;
    NPF_MPLS_DSCP_EXP_Type_t   type;
    NPF_MPLS_DSCP_EXP_Param_t  dscpExpEntry;
} NPF_MPLS_DSCP_EXP_EntryResp_t;
```

5.11.6 NPF_MPLS_NHFLE_CreateResp_t

This structure type definition holds response/return information provided by the API implementation during an NHLFE creation callback.

```
typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_NHFLE_Handle_t    nhlfeHandle;
   NPF_MPLS_DSCP_EXP_Handle_t dscpExpHandle;
} NPF_MPLS_NHFLE_CreateResp_t;
```

5.11.7 NPF_MPLS_NHLFE_EntryResp_t

This structure type definition holds response/return information provided by the API implementation during NHLFE API callbacks.

```
typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_NHLFE_Handle_t    nhlfeHandle;
   NPF_MPLS_NHLFE_Param_t     nhlfeEntry;
   NPF_MPLS_DSCP_EXP_Handle_t dscpExpHandle;
} NPF_MPLS_NHLFE_EntryResp_t;
```

5.11.8 NPF_MPLS_NHLFE_SET_CreateResp_t

This structure type definition holds response/return information provided by the API implementation during an NHLFE SET creation callback.

```
typedef struct {
   NPF_uint32_t                arrayIndex;
   NPF_MPLS_ReturnCode_t       returnCode;
   NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;
   NPF_uint32_t                numNhlfeResp;
   NPF_MPLS_NHLFE_CreateResp_t **numNhlfeResp;
} NPF_MPLS_NHLFE_SET_CreateResp_t;
```

5.11.9 NPF_MPLS_NHLFE_SET_EntryResp_t

This structure type definition holds response/return information provided by the API implementation during NHLFE SET API callbacks.

```
typedef struct {
   NPF_uint32_t                arrayIndex;
   NPF_MPLS_ReturnCode_t       returnCode;
   NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;
   NPF_MPLS_NHLFE_SET_Param_t  nhlfeEntry;
} NPF_MPLS_NHLFE_SET_EntryResp_t;
```

5.11.10 NPF_MPLS_AsyncResponse_t

This structure type definition holds asynchronous response/return information provided by the MPLS API implementation for an MPLS API during a callback.

```
typedef struct {
   NPF_MPLS_ReturnCode_t returnCode; /* Return code for the call */
   union {
      NPF_MPLS_LSP_CreateResp_t       lspCreateResp;
      NPF_MPLS_LSP_EntryResp_t        lspEntryResp;
      NPF_MPLS_LSP_Handle_t           lspHandle;
      NPF_MPLS_LSP_Stats_t            lspStatsResp;
      NPF_MPLS_NHLFE_CreateResp_t     nhlfeCreateResp;
      NPF_MPLS_NHLFE_EntryResp_t      nhlfeEntryResp;
      NPF_MPLS_NHLFE_Handle_t         nhlfeHandle;
      NPF_MPLS_NHLFE_Stats_t          nhlfeStatsResp;
      NPF_MPLS_NHLFE_SET_Handle_t     nhlfeSetHandle;
      NPF_MPLS_NHLFE_SET_CreateResp_t nhlfeSetCreateResp;
      NPF_MPLS_NHLFE_SET_EntryResp_t  nhlfeSetEntryResp;
      NPF_MPLS_NHLFE_SET_Stats_t      nhlfeSetStatsResp;
      NPF_MPLS_DSCP_EXP_CreateResp_t  dscpExpCreateResp;
      NPF_MPLS_DSCP_EXP_EntryResp_t   dscpExpEntryResp;
      NPF_MPLS_DSCP_EXP_Stats_t       dscpExpStats;
      NPF_MPLS_DSCP_EXP_Handle_t      dscpExpHandle;
      NPF_uint32_t                    tableSpaceRemaining;
      NPF_uint32_t                    unused;
   } u;
} NPF_MPLS_AsyncResponse_t;
```

Note: This structure type contains a return code for the call and an embedded return code for each structure within it. The embedded return codes are used for further investigation of the cause of the error (for debugging purposes). For example, if the `NPF_MPLS_AsyncResponse_t` structure was a response for a failed LSP create (where we are creating the LSP from scratch i.e. full NHLFE, NHLFE SET and ILM params), then the `returnCode` in the `NPF_MPLS_AsyncResponse_t` structure can be `NPF_MPLS_E_INVALID_LSP_PARAM` where further investigation of the `NPF_MPLS_AsyncResponse_t` structure can lead to which LSP param failed (an error in the NHLFE, ILM, …). The `NPF_MPLS_LSP_CreateResp_t` structure contains an `NPF_MPLS_NHLFE_SET_CreateResp_t` structure that contains an array of `NPF_MPLS_NHLFE_CreateResp_t` structures which in turn contain an error code for the cause of a particular NHLFE entry error (the NHLFE causing the error can now be identified as the cause of the bundled LSP create error and further debugged if needed).

5.11.11 NPF_MPLS_CallbackData_t

This structure type definition holds the call back data information.

```
typedef struct {
   NPF_MPLS_CallbackType_t  type;    /* Callback function type            */
   NPF_boolean_t            allOk;   /* TRUE if all functions completed OK */
   NPF_uint32_t             numResp; /* Number of responses in array       */
   NPF_MPLS_AsyncResponse_t *resp;   /* Pointer to response structures     */
} NPF_MPLS_CallbackData_t;
```

CALL BACK TYPE	Field associated
NPF_MPLS_LSP_ENTRY_CREATE	LspCreateResp
NPF_MPLS_LSP_ENTRY_DELETE	LspHandle
NPF_MPLS_LSP_ENTRY_MODIFY	LspHandle
NPF_MPLS_LSP_ATTRIBUTE_QUERY	TableSpaceRemaining
NPF_MPLS_LSP_ENTRY_QUERY	LspEntryResp
NPF_MPLS_LSP_ENTRY_STATS_QUERY	LspStatsResp
NPF_MPLS_NHLFE_ENTRY_CREATE	NhlfeCreateResp
NPF_MPLS_NHLFE_ENTRY_DELETE	NhlfeHandle
NPF_MPLS_NHLFE_ENTRY_MODIFY	NhlfeHandle
NPF_MPLS_NHLFE_ATTRIBUTE_QUERY	TableSpaceRemaining
NPF_MPLS_NHLFE_ENTRY_QUERY	NhlfeEntryResp
NPF_MPLS_NHLFE_STATS_QUERY	NhlfeStatsResp
NPF_MPLS_NHLFE_SET_CREATE	NhlfeSetCreateResp
NPF_MPLS_NHLFE_SET_DELETE	NhlfeSetHandle
NPF_MPLS_NHLFE_SET_MODIFY	NhlfeSetHandle
NPF_MPLS_NHLFE_SET_ATTRIBUTE_QUERY	TableSpaceRemaining
NPF_MPLS_NHLFE_SET_ENTRY_QUERY	NhlfeSetEntryResp
NPF_MPLS_NHLFE_SET_STATS_QUERY	NhlfeSetStatsResp
NPF_MPLS_DSCPEXP_ENTRY_CREATE	DscpExpCreateResp
NPF_MPLS_ DSCPEXP_ENTRY_DELETE	DscpExpHandle
NPF_MPLS_ DSCPEXP_ENTRY_MODIFY	DscpExpHandle
NPF_MPLS_ DSCPEXP_ATTRIBUTE_QUERY	TableSpaceRemaining
NPF_MPLS_ DSCPEXP_ENTRY_QUERY	DscpExpEntryResp
NPF_MPLS_ DSCPEXP_STATS_QUERY	DscpExpStats

5.12 Error Codes

The following are asynchronous error codes returned in function callbacks.

```
#define NPF_MPLS_E_ALREADY_REGISTERED           (NPF_MPLS_BASE_ERR+1)
```

```
#define NPF_MPLS_E_BAD_CALLBACK_HANDLE          (NPF_MPLS_BASE_ERR+2)
#define NPF_MPLS_E_BAD_CALLBACK_FUNCTION        (NPF_MPLS_BASE_ERR+3)
#define NPF_MPLS_E_INVALID_LSP_PARAM            (NPF_MPLS_BASE_ERR+4)
#define NPF_MPLS_E_INVALID_LSP_HANDLE           (NPF_MPLS_BASE_ERR+5)
#define NPF_MPLS_E_INVALID_LSP_TYPE             (NPF_MPLS_BASE_ERR+6)
#define NPF_MPLS_E_INVALID_NHLFE_PARAM          (NPF_MPLS_BASE_ERR+7)
#define NPF_MPLS_E_INVALID_NHLFE_HANDLE         (NPF_MPLS_BASE_ERR+8)
#define NPF_MPLS_E_INVALID_NHLFESET_PARAM       (NPF_MPLS_BASE_ERR+9)
#define NPF_MPLS_E_INVALID_NHLFESET_HANDLE      (NPF_MPLS_BASE_ERR+10)
#define NPF_MPLS_E_INVALID_FEC                  (NPF_MPLS_BASE_ERR+11)
#define NPF_MPLS_E_INVALID_IN_LABEL             (NPF_MPLS_BASE_ERR+12)
#define NPF_MPLS_E_INVALID_OUT_LABEL            (NPF_MPLS_BASE_ERR+13)
#define NPF_MPLS_E_INVALID_LABEL_STACK          (NPF_MPLS_BASE_ERR+14)
#define NPF_MPLS_E_INVALID_NEXT_HOP_IP          (NPF_MPLS_BASE_ERR+15)
#define NPF_MPLS_E_INVALID_NEXT_HOP_L2MEDIA     (NPF_MPLS_BASE_ERR+16)
#define NPF_MPLS_E_INVALID_NHLFE_FWD_POLICY     (NPF_MPLS_BASE_ERR+17)
#define NPF_MPLS_E_INVALID_INTERFACE            (NPF_MPLS_BASE_ERR+18)
#define NPF_MPLS_E_UNKNOWN                      (NPF_MPLS_BASE_ERR+19)
#define NPF_MPLS_E_INVALID_DSCPEXP_HANDLE       (NPF_MPLS_BASE_ERR+20)
#define NPF_MPLS_E_ENTRY_ALREADY_EXIST          (NPF_MPLS_BASE_ERR+21)
#define NPF_MPLS_E_INSUFFICIENT_STORAGE         (NPF_MPLS_BASE_ERR+22)
#define NPF_MPLS_E_FUNCTION_NOT_SUPPORTED       (NPF_MPLS_BASE_ERR+23)
```

5.13 Data Structures for Event Notifications

All MPLS events are optional.

Note: Even if an implementation does not generate any of these events, it still needs to implement the event register and deregister function for interoperability.

5.13.1 NPF_MPLS_EventType_t

This enumerated type definition specifies the different MPLS events that can be generated by the MPLS API implementation.

```
typedef enum {
   NPF_MPLS_EV_ILM_MISS  = 1,   /* No ILM entry for label */
   NPF_MPLS_EV_ILM_NHLFE = 2,   /* Packet matches ILM without NHLFE */
   NPF_MPLS_EV_FTN_NHLFE = 3,   /* Packet matches FTN without NHLFE */
   NPF_MPLS_EV_NHLFE_MTU = 4,   /* Labeled packet exceeds MTU */
   NPF_MPLS_EV_NHLFE_L2  = 5,   /* Need next hop resolution */
   NPF_MPLS_EV_PKT_TTL   = 6,   /* Exceeded TTL */
   NPF_MPLS_EV_NHLFE_MISS_EVENT = 7 /* When NHLFE/NHLFE SET does not exist */
} NPF_MPLS_EventType_t;
```

The event data always includes the offending packet. It also specifies the event type, which identifies the event, and indicates how to interpret the entity identifier and locate the associated entity (if relevant). The combination of these three items (event type, associated entity and packet) is sufficient to handle the event.

Event	Associated Entity
NPF_MPLS_EV_ILM_MISS	Ingress Interface
NPF_MPLS_EV_ILM_NHLFE	LSP Entry
NPF_MPLS_EV_FTN_NHLFE	LSP Entry

`NPF_MPLS_EV_NHLFE_MTU`	LSP Entry
`NPF_MPLS_EV_NHLFE_L2`	NHLFE
`NPF_MPLS_EV_PKT_TTL`	LSP Entry
`NPF_MPLS_EV_NHLFE_MISS`	LSP Entry

NOTE: The frequency `NPF_MPLS_EV_ILM_MISS` and `NPF_MPLS_EV_NHLFE_MISS` generation is implementation dependent. Since the event can be generated per packet, which will be an overhead to the processor, the implementation can generate the MISS events per specified time interval or rate limit for a group of packets.

5.13.2 NPF_MPLS_EventData_t

This structure type definition holds information associated with an event, generated by the MPLS API implementation.

```
typedef struct {
   NPF_MPLS_Event_t eventType;  /* Event type */
   union {
      NPF_MPLS_LSP_Handle_t   lspHandle;
      NPF_MPLS_NHLFE_Handle_t nhlfeHandle;
   } u;
   NPF_IfHandle_t ingressInterface;
   NPF_uint32_t   packetLength; /* Length of packet   */
   void           *packetData;  /* Location of packet */
} NPF_MPLS_EventData_t;
```

5.13.3 NPF_MPLS_EventArray_t

This structure type definition holds an array of MPLS Event information.

```
typedef struct    {
   NPF_uint16_t         nData;      /* Number of events in array    */
   NPF_MPLS_EventData_t *eventData; /* Array of event notifications */
} NPF_MPLS_EventArray_t;
```

6 Function Calls

6.1 Completion Callback Function Calls

This callback function is for the application to register an asynchronous response handling routine to the MPLS API implementation. This callback function is to be implemented by the application, and to be registered to the MPLS API implementation through the NPF_MPLS_Register function.

For more information regarding the design and usage of completion callbacks, please refer to Section 7, "Function Invocation Model, Events and Completion Callbacks", of the Network Processing Forum Software API Conventions Implementation Agreement [2].

6.1.1 NPF_MPLS_CallbackFunc_t

Syntax

```
typedef void (*NPF_MPLS_CallbackFunc_t) (
         NPF_IN NPF_userContext_t userContext,
         NPF_IN NPF_correlator_t correlator,
         NPF_IN NPF_MPLS_CallbackData_t *callbackData);
```

Description

This function is a registered completion callback routine for handling MPLS asynchronous responses.

This is a required function.

Input Arguments

- userContext - The context item that was supplied by the application when the completion callback function was registered.
- correlator - The correlator item (or call ID) that was supplied by the application when the MPLS API function call was invoked.
- callbackData - Pointer to a structure containing an array of response information related to the particular MPLS API function call, which is identified by the type field in the call back data.

Output Arguments

None

Return Values

None

6.2 Event Notification Function Calls

This event notification function is for the application to register an event handler routine to the MPLS API implementation. This handler function is intended to be implemented by the application, and to be registered to the MPLS API implementation through the NPF_MPLS_EventRegister function.

6.2.1 NPF_MPLS_EventCallFunc_t

Syntax

```
typedef void (*NPF_MPLS_EventCallFunc_t) (
         NPF_IN NPF_userContext_t     userContext,
         NPF_IN NPF_MPLS_EventArray_t mplsEventArray);
```

Description

This function is registered event notification routine for handling MPLS events. One or more events can be notified to the application through a single invocation of this event handler function. Information on each event is represented in an array in the mplsEventArray structure, where the application can traverse through the array and process each of the events.

This is a required function. This function may be called any time after NPF_MPLS_EventRegister() is called for it.

Input Arguments

- userContext -The context item that was supplied by the application when the event handler function was registered.
- mplsEventArray **-**Data structure that contains an array of event information. See NPF_MPLS_EventArray_t definition for details.

Output Arguments

None

Return Values

None

6.3 Callback Registration/Deregistration Function Calls

This section defines the registration and de-registration functions used to install and remove an asynchronous response callback routine.

6.3.1 NPF_MPLS_Register

Syntax

```
NPF_error_t NPF_MPLS_Register(
         NPF_IN NPF_userContext_t         userContext,
         NPF_IN NPF_MPLS_FwCallbackFunc_t callbackFunc,
         NPF_OUT NPF_callbackHandle_t     *callbackHandle);
```

Description

This function is used by an application to register its completion callback function for receiving asynchronous responses related to MPLS API function calls. Applications MAY register multiple callback functions using this function. The callback function is identified by the pair of userContext and callbackFunc, and for each individual pair, a unique callbackHandle will be assigned for future reference.

Since the callback function is identified by both userContext and callbackFunc, duplicate registration of the same callback function with different userContext is allowed. In addition, the same userContext can be shared among different callback functions. Duplicate registration of the same userContext and callbackFunc pair has no effect, will output a handle that is already assigned to the pair, and will return `NPF_MPLS_E_ALREADY_REGISTERED`.

This is a required function.

Input Arguments

- userContext - A context item for uniquely identifying the context of the application registering the completion callback function. The exact value will be provided back to the registered completion callback function as its first parameter when it is called. Application can assign any value to the userContext and the value is completely opaque to the API implementation.

- callbackFunc - Pointer to the completion callback function to be registered.

Output Arguments

- callbackHandle - A unique identifier assigned for the registered userContext and callbackFunc pair. This handle will be used by the application to specify which callback to be called when invoking asynchronous API functions. It will also be used when de-registering the userContext and callbackFunc pair.

Return Values

- NPF_NO_ERROR- The registration completed successfully.
- NPF_MPLS_E_BAD_CALLBACK_FUNCTION – The callback function is NULL, or otherwise invalid.
- NPF_MPLS_E_ALREADY_REGISTERED- No new registration was made since the userContext and callback Function pair was already registered.

Notes

- This API function MUST be invoked by any application interested in receiving asynchronous responses for MPLS API function calls.
- This function operates in a synchronous manner, providing a return value as listed above.

6.3.2 NPF_MPLS_Deregister

Syntax

```
NPF_error_t NPF_MPLS_Deregister(
        NPF_IN NPF_callbackHandle_t callbackHandle);
```

Description

This function is used by an application to de-register a completion callback function, which was previously registered to handle asynchronous callbacks related to MPLS API invocations.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier returned to the application when the completion callback routine was registered. It represents a unique user context and callback function pair.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The de-registration completed successfully.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The de-registration did not complete successfully due to problems with the callback handle provided.

Notes

- This API function may be invoked by any application no longer interested in receiving asynchronous responses for MPLS API function calls.
- This function operates in a synchronous manner, providing a return value as listed above.

- There may be a timing window where outstanding callbacks continue to be delivered to the callback routine after the de-registration function has been invoked. It is the implementation's responsibility to guarantee that the callback function is not called after the deregister function has returned.

6.4 Event Registration/Deregistration Function Calls

This section defines the registration and de-registration functions used to install and remove an event handler routine.

6.4.1 NPF_MPLS_EventRegister

Syntax

```
NPF_error_t NPF_MPLS_EventRegister(
        NPF_IN  NPF_userContext_t           userContext,
        NPF_IN  NPF_MPLS_EventHandlerFunc_t   eventCallFunc,
        NPF_OUT NPF_callbackHandle_t         *eventCallHandle);
```

Description

This function is used by an application to register its event handling routine for receiving notifications of MPLS events. Applications MAY register multiple event handling routines using this function. The event handling routine is identified by the pair of userContext and eventCallFunc, and for each individual pair, a unique eventCallHandle will be assigned for future reference.

Since the event handling routine is identified by both userContext and eventCallFunc, duplicate registration of same event handling routine with different userContext is allowed. In addition, same userContext can be shared among different event handling routines. Duplicate registration of the same userContext and eventCallFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_ALREADY_REGISTERED.

This is a required function.

Input Arguments

- userContext - A context item for uniquely identifying the context of the application registering the event handler function. The exact value will be provided back to the registered event handler function as its first parameter when it is called. Application can assign any value to the userContext and the value is completely opaque to the API implementation.
- eventCallFunc - Pointer to the event handler function to be registered.

Output Arguments

- eventCallHandle - A unique identifier assigned for the registered userContext and eventCallFunc pair. This handle will be used by the application de-registering the userContext and mplsEventHandlerFunc pair.

Return Values

- NPF_NO_ERROR - The registration completed successfully.
- NPF_MPLS_E_BAD_CALLBACK_FUNCTION - mplsEventHandlerFunc is NULL or not recognized.
- NPF_MPLS_E_ALREADY_REGISTERED - No new registration was made since the userContext and mplsEventHandlerFunc pair was already registered.

Notes

- This API function may be invoked by any application interested in receiving MPLS events.
- This function operates in a synchronous manner, providing a return value as listed above.
- Even if an implementation does not support events, the implementation needs to implement the function to enable interoperability.

6.4.2 NPF_MPLS_EventDeregister

Syntax

```
NPF_error_t NPF_MPLS_EventDeregister(
         NPF_IN NPF_callbackHandle_t eventCallHandle);
```

Description

This function is used by an application to de-register an event handling routing, which was previously registered to receive notification of MPLS events. It represents a unique user context and event handling routine pair.

This is a required function.

Input Arguments

- eventCallHandle - The unique identifier representing the pair of user context and event handler function to be de-registered.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The de-registration completed successfully.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The deregistration did not complete successfully due to problems with the callback handle provided.

Notes

- This API function may be invoked by any application no longer interested in receiving MPLS events.
- This function operates in a synchronous manner, providing a return value as listed above.
- There may be a timing window where outstanding events continue to be delivered to the event routine after the de-registration function has been invoked. It is the implementation's responsibility to guarantee that the event handling function is not called after the deregister function has returned.
- Even if an implementation does not support events, the implementation needs to implement the function to enable interoperability.

6.5 MPLS Forwarding Service API

6.5.1 NPF_MPLS_LSP_EntryCreate

Syntax

```
NPF_error_t  NPF_MPLS_LSP_EntryCreate(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          nMplsLsp,
         NPF_IN NPF_MPLS_LSP_t        **mplsLspArray);
```

Description

This function creates one or more MPLS LSP entries. The callback function will receive as many handles as NPF_MPLS_LSP_EntryCreate() could successfully create, and error codes for the rest. NPF_MPLS_LSP_EntryCreate function call creates and updates the NHLFE Set and NHLFE related information in case when the NHLFE set and NHLFE were not created prior to the LSP creation.

If one of the embedded structures fails, the LSP entry creation should fail without partial installation.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nMplsLsp - Number of elements in the mplsLspArray, i.e. number of LSP entries to be created.
- mplsLspArray - An array of LSP information pointers that are to be used when creating the LSP entries.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_INVALID_LSP_PARAM – The LSPs were not created due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nMplsLsp asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of NPF_MPLS_LSP_ENTRY_CREATE type. Each response contains one or more LSP handles that uniquely identify the LSP entry or a possible error code. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.

- NPF_MPLS_E_INSUFFICIENT_STORAGE - LSP entry could not be created due to lack of memory/space.
- NPF_MPLS_E_INVALID_NHLFESET_PARAM - LSP entry could not be created due to problem with NHLFE Set Param.
- NPF_MPLS_E_INVALID_NHLFESET_HANDLE - LSP entry could not be created due to problem with NHLFE Set handle.
- NPF_MPLS_E_INVALID_NHLFE_PARAM - LSP entry could not be created due to problem with NHLFE Param.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE - LSP entry could not be created due to problem with NHLFE handle.
- NPF_MPLS_E_INVALID_IN_LABEL - LSP entry could not be created due to problem with the incoming label value.
- NPF_MPLS_E_INVALID_LSP_TYPE - LSP entry could not be created due to problem with LSP type.
- NPF_MPLS_E_INLVALD_OUT_LABEL - LSP entry could not be created due to problem with the outgoing label value(s) in the label stack.
- NPF_MPLS_E_INVALID_NEXT_HOP_IP - LSP entry could not be created due to problem with the next hop IP address.
- NPF_MPLS_E_INVALID_NHLFE_FWD_POLICY – LSP entry could not be created due to problem with NHLFE forwarding policy.
- NPF_MPLS_E_ENTRY_ALREADY_EXIST – LSP entry to be added already exists.

Notes

When determining whether an LSP entry has been created, the parameters that uniquely identify an LSP entry based on the LSP type will be used. For an ILM type LSP entry the Incoming label and incoming interface will be used for identification. For a tunnel type LSP entry, the LSP ID will be used for identification.

6.5.2 NPF_MPLS_LSP_EntryDelete

Syntax

```
NPF_error_t  NPF_MPLS_LSP_EntryDelete(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          nHandles,
         NPF_IN NPF_MPLS_LSP_Handle_t *mplsLspHandleArray);
```

Description

This function deletes one or more MPLS LSP Entries. This function call will not delete the NHLFE Set or NHLFEs associated with the LSP entry. NHLFE Set or NHLFE deletion is to be done explicitly with the function calls associated with NHLFE Set and NHLFE.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of LSP entries to be deleted.
- mplsLspHandleArray - Pointer to an array of handles of the LSP entries to be deleted.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The LSP entries were not deleted due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_LSP_ENTRY_DELETE. Each response contains the handle of the deleted LSP, or a possible error code and the handle of the LSP entry that is to be deleted. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.
- NPF_MPLS_E_INVALID_LSP_HANDLE – LSP entry could not be deleted due to problem with the LSP handle.

Notes

None

6.5.3 NPF_MPLS_LSP_EntryModify

Syntax

```
NPF_error_t  NPF_MPLS_LSP_EntryModify(
         NPF_IN NPF_callbackHandle_t   callbackHandle,
         NPF_IN NPF_correlator_t       correlator,
         NPF_IN NPF_errorReporting_t   errorReporting,
         NPF_IN NPF_uint32_t           nHandles,
         NPF_IN NPF_MPLS_LSP_Handle_t  *mplsLspHandleArray,
         NPF_IN NPF_MPLS_LSP_t         *mplsLspArray);
```

Description

This function modifies the LSP entry information. For a given ILM entry the associated NHLFE Set information is updated.

This function call does not create or modify the information of an NHLFE Set or an NHLFE. NHLFE Set or NHLFE creation is to be done with the APIs associated with NHLFE Set and NHLFE or as part of LSP entry creation. NHLFE Set or NHLFE modification is to be done with the APIs associated with NHLFE Set and NHLFE.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of LSP entries to be modified.
- mplsLspHandleArray - Pointer to an array of handles of the LSP entries to be modified.
- mplsLspArray - Pointer to an array LSP entry information associated with the handles to be modified.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The LSP entries were not modified due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_LSP_ENTRY_MODIFY. Each response contains the handle of the modified LSP entry, or a possible error code and the handle of the LSP entry that was to be modified. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_LSP_HANDLE – LSP entry could not be modified due to invalid LSP handle.
- NPF_MPLS_E_INVALID_NHLFESET_HANDLE - LSP entry could not be modified due to invalid NHLFESET handle.

Notes

None

6.5.4 NPF_MPLS_LSP_EntryQuery

Syntax

```
NPF_error_t  NPF_MPLS_LSP_EntryQuery(
         NPF_IN NPF_callbackHandle_t callbackHandle,
         NPF_IN NPF_correlator_t     correlator,
         NPF_IN NPF_errorReporting_t errorReporting,
         NPF_IN NPF_uint32_t         numInfo,
         NPF_IN NPF_MPLS_LSP_Info_t  *mplsLspInfoArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS LSP entry response structures (NPF_MPLS_LSP_EntryResp_t) containing the settings and handle for a specified LSP entry. The LSP entry may be specified by either the LSP Handle or the key values, which make the LSP unique. In either case, the full LSP entry will be returned from the SAPI.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- numInfo - The number of LSP entries to retrieve.
- mplsLspInfoArray - Pointer to an array of LSP entry Information structures.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_E_UNKNOWN – The LSPs were not queried due to problems encountered when handling the input parameters.
- NPF_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of numInfo asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_LSP_ENTRY_QUERY. Each successful response contains the pointer to the MPLS LSP entry structure and the LSP Handle or a possible error code. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_HANDLE – An NPF_MPLS_LSP_Handle_t is null or invalid.
- NPF_MPLS_E_UNKNOWN – The LSP entry could not be located based on the unique key values.

Notes

None

6.5.5 NPF_MPLS_LSP_StatsGet

Syntax

```
NPF_error_t  NPF_MPLS_LSP_StatsGet(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          nHandles,
         NPF_IN NPF_MPLS_LSP_Handle_t *mplsLspHandleArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS LSP entry statistics structures (NPF_MPLS_LSP_Statistics_t) containing the current counter values for one or more indicated LSP entry.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of LSP entries to get statistics for.
- mplsLspHandleArray - Pointer to an array of LSP entry handles.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The LSP entries statistics were not obtained due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_LSP_ENTRY_STATS_QUERY. Each successful response contains the pointer to the MPLS LSP statistics structures or a possible error code. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_LSP_HANDLE – An NPF_MPLS_LSP_Handle is null or invalid.

Notes

None

6.5.6 NPF_MPLS_LSP_AttributeQuery

Syntax

```
NPF_error_t NPF_MPLS_LSP_AttributeQuery(
         NPF_IN NPF_callbackHandle_t   callbackHandle,
         NPF_IN NPF_correlator_t       correlator,
         NPF_IN NPF_errorReporting_t   errorReporting);
```

Description

This call will provide information about the characteristics of the LSP table/database. Currently, the attributes available are:

An estimate of how many free entries are in this table/database.

This is an optional function. Implementations that do not support queries MUST implement a stub of this function and MUST either immediately return NPF_MPLS_FUNCTION_NOT_SUPPORTED when called or MUST return NPF_MPLS_FUNCTION_NOT_SUPPORTED in the returnCode field of the asynchronous callback structure.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_E_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the LSP table/database. The callback structure returned is of type NPF_MPLS_LSP_ATTRIBUTE_QUERY. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.
- NPF_MPLS_E_FUNCTION_NOT_SUPPORTED – The attribute query function for the LSP entry table is not supported by this implementation.

Notes

- Applications may use this query API function to obtain information useful in maintaining the LSP table/database. For example, prior to creating new LSP entries, the application might query the available free space of the LSP table/database and, therefore, be able to know when it cannot add any more LSP entries.

- The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many "Create" requests will succeed.
- The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

6.5.7 NPF_MPLS_DSCP_EXP_Create

Syntax

```
NPF_error_t NPF_MPLS_DSCP_EXP_Create(
        NPF_IN NPF_callbackHandle_t      callbackHandle,
        NPF_IN NPF_correlator_t          correlator,
        NPF_IN NPF_errorReporting_t      errorReporting,
        NPF_IN NPF_MPLS_DSCP_EXP_Type_t  type,
        NPF_IN NPF_uint32_t              nTables,
        NPF_IN NPF_MPLS_DSCP_EXP_Param_t **dscpExptablesArray);
```

Description

This function creates one or more DSCP to EXP or EXP to DSCP tables. The callback function will receive as many handles as NPF_MPLS_DSCP_EXP_Create() could successfully create, and error codes for the rest. If a table id already exists an error is returned.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- type - Either NPF_MPLS_DSCPEXP_DTOE or NPF_MPLS_DSCPEXP_ETOD.
- nTables - Number of tables to create.
- dscpExptablesArray - An array of DSCPEXP tables.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The LSPs were not created due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nTables asynchronous responses (NPF_MplsAsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_DSCPEXP_ENTRY_CREATE. Each response contains one or more table handles, embedded in NPF_MPLS_DSCP_EXP_CreateResp_t structures, that uniquely identifies the table or a possible error code. Possible return codes are

- NPF_NO_ERROR - Operation successful.
- NPF_MPLS_E_ENTRY_ALREADY_EXIST - Table already exist.
- NPF_MPLS_E_INSUFFICIENT_STORAGE - Table could not be created due to lack of memory/space.

Notes

None

6.5.8 NPF_MPLS_DSCP_EXP_Delete

Syntax

```
NPF_error_t  NPF_MPLS_DSCP_EXP_Delete(
        NPF_IN NPF_callbackHandle_t       callbackHandle,
        NPF_IN NPF_correlator_t           correlator,
        NPF_IN NPF_errorReporting_t       errorReporting,
        NPF_IN NPF_uint32_t               nHandles,
        NPF_IN NPF_MPLS_DSCP_EXP_Handle_t *dscpExpHandleArray);
```

Description

This function deletes one or more DSCPEXP tables. The tables indicated by the nHandles will be removed. When a table is removed, there is no predefined behavior except for Best Effort.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of tables to be deleted.
- dscpExpHandleArray - Pointer to an array of handles of the tables to be deleted.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The tables were not deleted due to problems encountered when handling the input parameters.

- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_DSCPEXP_ENTRY_DELETE. Each response contains the handle of the deleted table, or a possible error code and the handle of the table that is to be deleted. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.
- NPF_MPLS_E_INVALID_DSCPEXP_HANDLE – DSCPEXP table could not be deleted due to problem with the handle.

Notes

None

6.5.9 NPF_MPLS_DSCP_EXP_Modify

Syntax

```
NPF_error_t  NPF_MPLS_DSCP_EXP_Modify(
          NPF_IN NPF_callbackHandle_t       callbackHandle,
          NPF_IN NPF_correlator_t           correlator,
          NPF_IN NPF_errorReporting_t       errorReporting,
          NPF_IN NPF_uint32_t               nHandles,
          NPF_IN NPF_MPLS_DSCP_EXP_Handle_t *dscpExpHandleArray,
          NPF_IN NPF_MPLS_DSCP_EXP_Param_t  *dscpExpArray);
```

Description

This function modifies the tables pointed at by the passed handles to their corresponding dscpExpArray param entry. The dscpExpArray entries will override the existent tables.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of tables to be modified.
- dscpExpHandleArray - Pointer to an array of handles of the tables to be modified.
- dscpExpArray - Pointer to an array of tables associated with the table handles to be modified.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.

- NPF_MPLS_E_UNKNOWN – The tables were not modified due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_DSCPEXP_ENTRY_MODIFY. Each response contains the handle of the modified table, or a possible error code and the handle of the table that is to be modified. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_DSCPEXP_HANDLE - The DSCPEXP handle is not valid.

Notes

None

6.5.10 NPF_MPLS_DSCP_EXP_StatsGet

Syntax

```
NPF_error_t  NPF_MPLS_DSCP_EXP_StatsGet(
         NPF_IN NPF_callbackHandle_t       callbackHandle,
         NPF_IN NPF_correlator_t           correlator,
         NPF_IN NPF_errorReporting_t       errorReporting,
         NPF_IN NPF_uint32_t               nHandles,
         NPF_IN NPF_MPLS_DSCP_EXP_Handle_t *dscpExpHandleArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS DSCPEXP statistics structures(NPF_MPLS_DSCP_EXP_Stats_t) containing the current counter values for one or more indicated table information.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of tables to that needs to be queried.
- dscpExpHandleArray - Pointer to an array of handles of the tables to be queried.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.

- NPF_MPLS_E_UNKNOWN – The DSCP-EXP table entries statistics were not obtained due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_DSCPEXP_STATS_QUERY. Each response contains the NPF_MPLS_DSCP_EXP_Stats_t structure of the queried table, or a possible error code and the handle of the table that is to be modified. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_DSCPEXP_HANDLE - The DSCPEXP handle is not valid.

Notes

None

6.5.11 NPF_MPLS_DSCP_EXP_AttributeQuery

Syntax

```
NPF_error_t NPF_MPLS_DSCP_EXP_AttributeQuery(
          NPF_IN NPF_callbackHandle_t   callbackHandle,
          NPF_IN NPF_correlator_t       correlator,
          NPF_IN NPF_errorReporting_t   errorReporting);
```

Description

This call will provide information about the characteristics of the DSCPEXP table/database. Currently, the attributes available are:

- An estimate of how many free entries are in this table/database.

This is an optional function. Implementations that do not support queries MUST implement a stub of this function and MUST either immediately return NPF_MPLS_FUNCTION_NOT_SUPPORTED when called or MUST return NPF_MPLS_FUNCTION_NOT_SUPPORTED in the returnCode field of the asynchronous callback structure.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.

- NPF_MPLS_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the DSCPEXP table/database. The callback structure returned is of type NPF_MPLS_DSCPEXP_ATTRIBUTE_QUERY. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.
- NPF_MPLS_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the DSCPEXP table/database. For example, prior to creating new tables, the application might query the available free space of the table/database and, therefore, be able to know when it cannot add any more tables.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many “Create” requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

6.5.12 NPF_MPLS_DSCP_EXP_EntryQuery

Syntax

```
NPF_error_t  NPF_MPLS_DSCP_EXP_EntryQuery(
         NPF_IN NPF_callbackHandle_t callbackHandle,
         NPF_IN NPF_correlator_t     correlator,
         NPF_IN NPF_errorReporting_t errorReporting,
         NPF_IN NPF_uint32_t         numTables,
         NPF_IN NPF_MplsDscpExp_t    *dscpExpArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS DSCPEXP tables response structures (NPF_MplsDscpExpResp_t) containing the settings and handle for a specified DSCP EXP table. The table may be specified by either the table Handle or the table id, which make the table unique. In either case, the full DSCP EXP table will be returned from the SAPI.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.

- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- numTables - the number of tables to retrieve.
- dscpExpArray - pointer to an array of DSCPEXP tables.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The DSCP-EXP tables were not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of numTables asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_DSCPEXP_ENTRY_QUERY. Each successful response contains the DSCPEXP table structure and the table Handle embedded in a NPF_MPLS_DSCP_EXP_EntryResp_t structure or a possible error code. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_HANDLE – An NPF_MPLS_DSCP_EXP_Handle_t is null or invalid.

Notes

None

6.5.13 NPF_MPLS_NHLFE_Create

Syntax

```
NPF_error_t NPF_MPLS_NHLFE_Create(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          nNhlfe,
         NPF_IN NPF_MPLS_NHLFE_t      **mplsNhlfeArray);
```

Description

This function creates one or more MPLS NHLFEs. The callback function will receive as many handles as NPF_MPLS_NHLFE_Create() could successfully create, and error codes for the rest.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.

- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nNhlfe - Number of NHLFEs to create.
- mplsNhlfeArray - An array of NHLFE pointers.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFEs were not created due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nNhlfe asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_ENTRY_CREATE. Each response contains one or more NHLFE handles that uniquely identify the NHLFE or a possible error code. Possible return codes are

- NPF_NO_ERROR - operation successful.
- NPF_MPLS_E_INSUFFICIENT_STORAGE - NHLFE could not be created due to lack of memory/space.
- NPF_MPLS_E_INLVALD_OUT_LABEL - NHLFE entry could not be created due to problem with the outgoing label value(s) in the label stack.
- NPF_MPLS_E_INVALID_LABEL_STACK - NHLFE entry could not be created due to invalid value(s) in the label stack.
- NPF_MPLS_E_INVALID_NEXT_HOP_IP - NHLFE entry could not be created due to problem with the next hop IP address.
- NPF_MPLS_E_INVALID_INTERFACE - NHLFE entry could not be created due to invalid outgoing interface.
- NPF_MPLS_E_ENTRY_ALREADY_EXIST – NHLFE entry to be added already exists.

Notes

None

6.5.14 NPF_MPLS_NHLFE_Delete

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_Delete(
         NPF_IN NPF_callbackHandle_t    callbackHandle,
         NPF_IN NPF_correlator_t        correlator,
         NPF_IN NPF_errorReporting_t    errorReporting,
         NPF_IN NPF_uint32_t            nHandles,
         NPF_IN NPF_MPLS_NHLFE_Handle_t *mplsNhlfeHandleArray);
```

Description

This function deletes one or more NHLFEs. If an entry exists in the NHLFE table/database as indicated by the NHLFE Handle, then it will be removed.

If an NHLFE is removed, a reference to the removed entry by the forwarding plane MAY generate an NPF_MPLS_EV_NHLFE_MISS event.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of NHLFEs to delete.
- mplsNhlfeHandleArray - Pointer to an array of handles of the NHLFEs to be deleted.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFEs were not deleted due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_ENTRY_DELETE. Each response contains the handle of the deleted NHLFE, or a possible error code and the handle of the NHLFE that is to be deleted. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE - NHLFE entry could not be deleted due to invalid NHLFE handle.

Notes

None

6.5.15 NPF_MPLS_NHLFE_Modify

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_Modify(
         NPF_IN NPF_callbackHandle_t    callbackHandle,
         NPF_IN NPF_correlator_t        correlator,
         NPF_IN NPF_errorReporting_t    errorReporting,
         NPF_IN NPF_uint32_t            nHandles,
         NPF_IN NPF_MPLS_NHLFE_Handle_t *mplsNhlfeHandleArray,
         NPF_IN NPF_MPLS_NHLFE_t        *mplsNhlfeArray);
```

Description

This function modifies the NHLFE information. For a given NHLFE entry the associated DSCPEXP table handle is updated (key fields can not be modified). This function cannot modify the content nor create the DSCPEXP table.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of NHLFEs to be modified.
- mplsNhlfeHandleArray - Pointer to an array of handles of the NHLFEs to be modified.
- mplsNhlfeArray - Pointer to an array NHLFE information associated with the handles to be modified.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFEs were not modified due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_ENTRY_MODIFY. Each response contains the handle of the modified LSP, or a possible error code and the handle of the LSP that was to be modified. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_DSCPEXP_HANDLE - NHLFE entry could not be modified due to problem with DSCPEXP handle.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE - NHLFE entry could not be modified due to problem with NHLFE handle.

Notes

None

6.5.16 NPF_MPLS_NHLFE_StatsGet

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_StatsGet(
         NPF_IN NPF_callbackHandle_t    callbackHandle,
         NPF_IN NPF_correlator_t        correlator,
         NPF_IN NPF_errorReporting_t    errorReporting,
         NPF_IN NPF_uint32_t            nHandles,
         NPF_IN NPF_MPLS_NHLFE_Handle_t *mplsNhlfeHandleArray);
```

Description

This function returns, via a callback, a pointer to one or more the MPLS NHLFE statistics structures(NPF_MPLS_NHLFE_Statistics_t)containing the current counter values for one or more indicated NHLFE information.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- Correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of NHLFEs to get statistics for.
- mplsNhlfeHandleArray - Pointer to an array of NHLFE handles.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFEs Statistics were not obtained due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_STATS_QUERY. Each successful response contains the pointer to the MPLS NHLFE statistics structures or an error code. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE – An NPF_MPLS_NHLFE_Handle is null or invalid.

Notes

None

6.5.17 NPF_MPLS_NHLFE_AttributeQuery

Syntax

```
NPF_error_t NPF_MPLS_NHLFE_AttributeQuery(
          NPF_IN NPF_callbackHandle_t   callbackHandle,
          NPF_IN NPF_correlator_t       correlator,
          NPF_IN NPF_errorReporting_t   errorReporting);
```

Description

This call will provide information about the characteristics of the NHLFE table/database. Currently, the attributes available are:

An estimate of how many free entries are in this table/database.

This is an optional function. Implementations that do not support queries MUST implement a stub of this function and MUST either immediately return NPF_MPLS_FUNCTION_NOT_SUPPORTED when called or MUST return NPF_MPLS_FUNCTION_NOT_SUPPORTED in the returnCode field of the asynchronous callback structure.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN - The table was not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_E_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the NHLFE table/database. The callback structure returned is of type NPF_MPLS_NHLFE_ATTRIBUTE_QUERY. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.
- NPF_MPLS_E_FUNCTION_NOT_SUPPORTED – The attribute query function for the address resolution table is not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the NHLFE table/database. For example, prior to creating new LSPs, the application might query the available free space of the NHLFE table/database and, therefore, be able to know when it cannot add any more NHLFEs.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many "Create" requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

6.5.18 NPF_MPLS_NHLFE_EntryQuery

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_EntryQuery(
         NPF_IN NPF_callbackHandle_t callbackHandle,
         NPF_IN NPF_correlator_t     correlator,
         NPF_IN NPF_errorReporting_t errorReporting,
         NPF_IN NPF_uint32_t         numNhlfe,
         NPF_IN NPF_MPLS_NHLFE_t     *mplsNhlfeArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS NHLFE entry response structures (NPF_MPLS_NHLFE_EntryResp_t) containing the settings and handle for a specified NHLFE. The NHLFE may be specified by either the NHLFE Handle or the key values, which make the NHLFE unique. In either case, the full NHLFE entry will be returned from the SAPI.

This is a required function.

Input Parameters

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- numNhlfe - The number of NHLFEs to retrieve.
- mplsNhlfeArray - Pointer to an array of NHLFE structures.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFE Entries were not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of numNhlfe asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type

NPF_MPLS_NHLFE_ENTRY_QUERY. Each successful response contains the pointer to the MPLS NHLFE entry structure and the NHLFE Handle or an error code. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE – An NPF_MPLS_NHLFE_Handle_t is null or invalid.
- NPF_MPLS_E_UNKNOWN – The NHLFE could not be located based on the unique key values.

Notes

None

6.5.19 NPF_MPLS_NHLFE_SET_Create

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_SET_Create(
        NPF_IN NPF_callbackHandle_t  callbackHandle,
        NPF_IN NPF_correlator_t      correlator,
        NPF_IN NPF_errorReporting_t  errorReporting,
        NPF_IN NPF_uint32_t          nNhlfeSet,
        NPF_IN NPF_MPLS_NHLFE_SET_t  **mplsNhlfeSetArray);
```

Description

This function creates one or more MPLS NHLFE SETs. The callback function will receive as many handles as NPF_MPLS_NHLFE_Create() could successfully create, and error codes for the rest. NPF_MPLS_NHLFE_SET_Create API creates the NHLFE related information in case when the NHLFE were not created prior to the NHLFE SET creation.

If an NHLFE Set has an array of NHLFEs, then all NHLFEs have to be successfully installed or else the NHLFE Set will not be created.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nNhlfe - Number of NHLFE SETs to create.
- mplsNhlfeSetArray - An array of NHLFE SET information pointers.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFE SETs were not created due to problems encountered when handling the input parameters.

- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nNhlfeSet asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_SET_CREATE. Each response contains one or more NHLFE SET handle that uniquely identifies the NHLFE SET or a possible error code. Possible return codes are

- NPF_NO_ERROR - Operation successful.
- NPF_MPLS_E_INSUFFICIENT_STORAGE - NHLFE could not be created due to lack of memory/space.
- NPF_MPLS_E_INLVALD_OUT_LABEL – NHLFE Set entry could not be created due to problem with the outgoing label value(s) in the label stack.
- NPF_MPLS_E_INVALID_LABEL_STACK - NHLFE Set entry could not be created due to invalid value(s) in the label stack.
- NPF_MPLS_E_INVALID_NEXT_HOP_IP - NHLFE Set entry could not be created due to problem with the next hop IP address.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE - NHLFE Set entry could not be created due to invalid NHLFE handle.

Notes

None

6.5.20 NPF_MPLS_NHLFE_SET_Delete

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_SET_Delete(
        NPF_IN NPF_callbackHandle_t        callbackHandle,
        NPF_IN NPF_correlator_t            correlator,
        NPF_IN NPF_errorReporting_t        errorReporting,
        NPF_IN NPF_uint32_t                nHandles,
        NPF_IN NPF_MPLS_NHLFE_SET_Handle_t *mplsNhlfeSetHandleArray);
```

Description

This function deletes one or more NHLFE Sets. If an entry exists in the NHLFE SET table/database as indicated by the NHLFE SET Handle, then it will be removed. This function call will not delete the NHLFEs associated with the NHLFE Set. NHLFE deletion is to be done explicitly with the function calls associated with NHLFE.

If an NHLFE SET is removed, a reference to the removed entry by the forwarding plane MAY generate an NPF_MPLS_EV_NHLFE_MISS_EVENT.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.

- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of NHLFE Sets to be deleted.
- MplsNhlfeSetHandleArray - Pointer to an array of handles of the NHLFE Sets to be deleted.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFE Sets were not deleted due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_SET_DELETE. Each response contains the handle of the deleted NHLFE SET, or a possible error code and the handle of the NHLFE SET that is to be deleted. The possible return codes are:

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFESET_HANDLE - NHLFE Set entry could not be deleted due to invalid NHLFE Set handle.

Notes

None

6.5.21 NPF_MPLS_NHLFE_SET_Modify

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_SET_Modify(
        NPF_IN NPF_callbackHandle_t        callbackHandle,
        NPF_IN NPF_correlator_t            correlator,
        NPF_IN NPF_errorReporting_t        errorReporting,
        NPF_IN NPF_uint32_t                nNhlfe,
        NPF_IN NPF_MPLS_NHLFE_SET_Handle_t *mplsNhlfeSetHandleArray,
        NPF_IN NPF_MPLS_NHLFE_SET_t        *mplsNhlfeSetArray);
```

Description

This function modifies the NHLFE SET information. For a given NHLFE SET the policy based forwarding information and the associated NHFLE information are updated.

This API does not create or modify the information of an NHLFE. NHLFE modification is to be done with the APIs associated with NHLFE.

This is a required function.

Input Parameters

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.

- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nNhlfe - The number of NHLFE Sets to be modified.
- mplsNhlfeSetHandleArray - Pointer to an array of handles of the NHLFE SETs to be modified.
- mplsNhlfeSetArray - Pointer to an array NHLFE Set pointers - information associated with the NHLFE handles to be modified.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFE SETs were not modified due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_SET_MODIFY. Each response contains the handle of the modified NHLFE SET, or a possible error code and the handle of the NHLFE SET that was to be modified. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFE_SET_ HANDLE - NHLFE Set entry could not be modified due to invalid NHLFE Set handle.
- NPF_MPLS_E_INVALID_NHLFE_HANDLE - NHLFE Set entry could not be modified due to invalid NHLFE handle.
- NPF_MPLS_E_INLVALD_NHLFE_FWD_POLICY - NHLFE Set entry could not be modified due to problem with NHLFE forwarding policy.

Notes

None

6.5.22 NPF_MPLS_NHLFE_SET_EntryQuery

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_SET_EntryQuery(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          numNhlfeSet,
         NPF_IN NPF_MPLS_NHLFE_SET_t  *nhlfeSetArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS NHLFE Set entry response structures (NPF_MPLS_NHLFE_SET_EntryResp_t) containing the settings and handle for a specified NHLFE Set. The NHLFE Set may be specified by either the NHLFE Set Handle or the key values, which make the NHLFE Set unique. In either case, the full NHLFE Set entry will be returned from the SAPI.

This is a required function.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- numNhlfeSet - The number of NHLFE Sets to retrieve.
- nhlfeSetArray - Pointer to an array of NHLFE Set structures.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN – The NHLFE SETs were not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of numNhlfeSet asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_SET_ENTRY_QUERY. Each successful response contains the pointer to the MPLS NHLFE Set entry structure and the NHLFE Set Handle. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFESET_HANDLE – An NPF_MPLS_NHLFESET_Handle_t is null or invalid.
- NPF_MPLS_E_UNKNOWN – The NHLFE Set could not be located based on the unique key values.

Notes

None

6.5.23 NPF_MPLS_NHLFE_SET_StatsGet

Syntax

```
NPF_error_t  NPF_MPLS_NHLFE_StatsGet(
        NPF_IN NPF_callbackHandle_t        callbackHandle,
        NPF_IN NPF_correlator_t            correlator,
        NPF_IN NPF_errorReporting_t        errorReporting,
        NPF_IN NPF_uint32_t                nHandles,
        NPF_IN NPF_MPLS_NHLFE_SET_Handle_t *mplsNhlfeSetHandleArray);
```

Description

This function returns, via a callback, a pointer to one or more MPLS NHLFE SET statistics structures (NPF_MPLS_NHLFE_SET_Statistics_t) containing the current counter values for one or more indicated NHLFE SET information. The NHLFE SET does not contain any counters, but rather the counters associated with the set of NHLFEs - bundled under the SET - will be returned.

Note: In case an NHLFE is used by multiple NHLFE SETs the counters returned are for the shared NHLFE meaning that this call will not separate the information relative to the queried NHLFE SET handle.

This is an Optional function, since NHLFE stats can be queried individually.

Input Arguments

- callbackHandles - The unique identifier provided to the application when the completion callback routine was registered..
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.
- nHandles - The number of NHLFE SETs to get statistics for.
- mplsNhlfeSetHandleArray - Pointer to an array of NHLFE SET handles.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress
- NPF_MPLS_E_UNKNOWN – The NHLFE SETs Statistics were not obtained due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.

Asynchronous Response

A total of nHandles asynchronous responses (NPF_MPLS_AsyncResponse_t) will be passed to the callback function, in one or more invocations. The callback structure returned is of type NPF_MPLS_NHLFE_SET_STATS_QUERY. Each successful response contains the pointer to the MPLS NHLFE SET statistics structures or an error code. Possible return codes are

- NPF_NO_ERROR – Operation successful.
- NPF_MPLS_E_INVALID_NHLFESET_HANDLE – An NPF_MPLS_NHLFE_SET Handle is null or invalid.

Notes

None

6.5.24 NPF_MPLS_NHLFE_SET_AttributeQuery

Syntax

```
NPF_error_t NPF_MPLS_NHLFE_SET_AttributeQuery(
         NPF_IN NPF_callbackHandle_t   callbackHandle,
         NPF_IN NPF_correlator_t       correlator,
         NPF_IN NPF_errorReporting_t   errorReporting);
```

Description

This call will provide information about the characteristics of the NHLFE SET table/database. Currently, the attributes available are:

- An estimate of how many free entries are in this table/database.

This is an optional function. Implementations that do not support queries MUST implement a stub of this function and MUST either immediately return NPF_MPLS_FUNCTION_NOT_SUPPORTED when called or MUST return NPF_MPLS_FUNCTION_NOT_SUPPORTED in the returnCode field of the asynchronous callback structure.

Input Arguments

- callbackHandle - The unique identifier provided to the application when the completion callback routine was registered.
- correlator - A unique application invocation value that will be supplied to the asynchronous completion callback routine.
- errorReporting - An indication of whether the application desires to receive an asynchronous completion callback for this API function invocation.

Output Arguments

None

Return Values

- NPF_NO_ERROR - The operation is in progress.
- NPF_MPLS_E_UNKNOWN - The NHLFE SET table was not queried due to problems encountered when handling the input parameters.
- NPF_MPLS_E_BAD_CALLBACK_HANDLE – The callback handle is not valid.
- NPF_MPLS_E_FUNCTION_NOT_SUPPORTED – The attribute query capability is not supported by this implementation.

Asynchronous Response

A return code will be returned asynchronously along with an approximation of the number of free entries left in the NHLFE SET table/database. The callback structure returned is of type NPF_MPLS_NHLFE_SET_ATTRIBUTE_QUERY. Possible return codes are

- NPF_NO_ERROR - The operation completed successfully.
- NPF_MPLS_E_FUNCTION_NOT_SUPPORTED – The attribute query function for the NHLFE Set table is not supported by this implementation.

Notes

Applications may use this query API function to obtain information useful in maintaining the NHLFE SET table/database. For example, prior to creating new NHLFE Sets, the application might query the available free space of the NHLFE SET table/database and, therefore, be able to know when it cannot add any more NHLFE SETs.

The implementation SHOULD be conservative in what it returns. In other words, the value should be the amount of free space under the worst-case conditions, so that the application can be assured that at least this many "Create" requests will succeed.

The errorReporting parameter, included for the sake of consistency, is ignored. This function generates an asynchronous response, regardless of the value given in the errorReporting parameter.

6.5.25 Recovery APIs

The recovery API function calls are yet to be determined and are subject for further studies and discussions with the Foundations Group.

7 References

[1] NP Forum - Software API Framework Lexicon Implementation Agreement Revision 1.0

[2] NP Forum – Software API Conventions Implementation Agreement Revision 2.0

[3] NP Forum – Software API Framework Implementation Agreement Revision 1.0

[4] NP Forum – Interface Management API Implementation Agreement Revision 1.0

[5] NP Forum - IPv4 Unicast forwarding API Implementation Agreement Revision 1.0

[6] Multiprotocol Label Switching Architecture (MPLS) RFC3031

[7] MPLS Label Stack encoding RFC 3032

[8] Requirements for Traffic engineering over MPLS RFC 2702

[9] Constraint-Based LSP Setup using LDP RFC 3212

[10] RSVP-TE: Extensions to RSVP for LSP Tunnels RFC 3209

[11] Multi-Protocol Label Switching (MPLS) Support of Differentiated Services – RFC 3270

[12] Definition of the Differentiated Services Field (DS Field) in IPv4 and IPv6 Headers – RFC 2474

[13] An Architecture for Differentiated Services – RFC 2475

[14] Per Hop Behavior Identification Codes – RFC 3140

Appendix A Header File: NPF_MPLS.h

```
/*
 * This header file defines typedefs, constants, and functions
 * that apply to the NPF  MPLS forwarding service API*/
 */


typedef enum {
   NPF_MPLS_IPV4 = 1,
   NPF_MPLS_IPV6 = 2
} NPF_MPLS_IP_Type_t;

typedef struct{
   NPF_MPLS_IP_Type_t   ipAddrType;
   union {
      NPF_IPv4Address_t ipv4DestHostAddr; /* IPv4 Host Address */
      NPF_IPv6Address_t ipv6DestHostAddr; /* IPv6 Host Address */
   } u;
} NPF_MPLS_HostAddr_t;

typedef enum {
   NPF_MPLS_LABEL_TYPE_GENERIC = 1,
   NPF_MPLS_LABEL_TYPE_ATM     = 2,
   NPF_MPLS_LABEL_TYPE_FR      = 3
} NPF_MPLS_LabelType_t;

typedef NPF_uint32_t NPF_MPLS_ShimLabel_t;


typedef NPF_VccAddr_t NPF_MPLS_ATM_Label_t

typedef enum {
   NPF_MPLS_DLCI_10 = 1,
   NPF_MPLS_DLCI_23 = 2
} NPF_MPLS_DLCI_Type_t;


typedef struct {
   NPF_MPLS_DLCI_Type dlciType; /* Length of the DLCI in bits */
   NPF_uint32_t dlci;           /* DLCI */
} NPF_MPLS_FR_Label_t;


typedef struct {
   NPF_MPLS_LabelType_t labelType;    /* Type of label      */
   union {
      NPF_MPLS_ShimLabel_t shimLabel; /* Generic label      */
      NPF_MPLS_ATM_Label_t atmLabel;  /* ATM label          */
      NPF_MPLS_FR_Label_t  frLabel;   /* Frame Relay Label */
   } u;
} NPF_MPLS_Label_t;


typedef struct {
    NPF_int32_t       numLabels;  /* Number of labels */
    NPF_MPLS_Label_t* labelStack; /* Stack of labels  */
} NPF_MPLS_LabelStack_t;

typedef struct {
    NPF_uint8_t dscp; /*DSCP value */
    NPF_uint8_t exp;  /*EXP value  */
} NPF_MPLS_DSCP_EXP_Entry_t;

typedef struct {
    NPF_uint32_t tableId;    /* unique id set by user */
    NPF_uint8_t  numEntries; /* Number of entries in this map */
    NPF_MPLS_DSCP_EXP_Entry_t *entries; /* DSCP to EXP entries array*/
} NPF_MPLS_DSCP_EXP_Param_t;

typedef NPF_uint32_t NPF_MPLS_DSCP_EXP_Handle_t;

typedef enum {
   NPF_MPLS_DSCPEXP_HANDLE = 1,
   NPF_MPLS_DSCPEXP_PARAMS = 2
} NPF_MPLS_DSCP_EXP_InfoType_t;

typedef enum {
   NPF_MPLS_DSCPEXP_DTOE = 1,
   NPF_MPLS_DSCPEXP_ETOD = 2
} NPF_MPLS_DSCP_EXP_Type_t;

typedef struct {
   NPF_MPLS_DSCP_EXP_Type_t      type;      /* DSCP to Exp or Exp to DSCP*/
   NPF_MPLS_DSCP_EXP_InfoType_t  paramType; /* Handle or map information */
   union {
      NPF_MPLS_DSCP_EXP_Handle_t  mapHandle; /* Map Handle */
      NPF_MPLS_DSCP_EXP_Param_t   mapParam;  /* Map Parameters */
   } u;
} NPF_MPLS_DSCP_EXP_t;

typedef NPF_uint16_t NPF_MPLS_PSC_ID_t;

typedef enum {
   NPF_MPLS_PSCID_BE = 0
} NPF_MPLS_PSC_ID_Value_t;

typedef enum {
   NPF_MPLS_DS_LSP_TYPE_NONE = 0, /* non-Diffserv type */
   NPF_MPLS_DS_LSP_TYPE_ELSP = 1, /* ELSP */
   NPF_MPLS_DS_LSP_TYPE_LLSP = 2  /* LLSP */
} NPF_MPLS_DS_LSP_Type_t;

typedef enum {
   NPF_MPLS_DS_LSP_MODEL_NONE      = 0, /* non-Diffserv model */
   NPF_MPLS_DS_LSP_MODEL_PIPE      = 1, /* Pipe */
   NPF_MPLS_DS_LSP_MODEL_SHORTPIPE = 2, /* Short-pipe */
   NPF_MPLS_DS_LSP_MODEL_UNIFORM   = 3  /* Uniform */
} NPF_MPLS_DS_LSP_Model_t;
```

```
typedef struct {
   NPF_uint32_t maxRate;       /*Max or Peak rate (bps)*/
   NPF_uint32_t meanRate;      /*Mean rate (bps)*/
   NPF_uint32_t maxBurstSize;  /*Max Burst size in bytes*/
   NPF_uint32_t meanBurstSize; /*Mean Burst size in bytes*/
   NPF_uint32_t exBurstSize;   /*Excess Burst size in bytes*/
   NPF_uint32_t frequency;     /*Frequency of token refresh*/
   NPF_uint8_t  weight;        /*Weight associated with tunnel*/
   NPF_uint8_t  trafficClass;  /*Derived from parameters above*/
} NPF_MPLS_TE_Param_t;

typedef NPF_uint32_t NPF_MPLS_NHLFE_Handle_t;

typedef struct {
   NPF_IfHandle_t        egressInterface; /* Outgoing interface */
   NPF_MPLS_HostAddr_t   nextHopAddr;     /* Next Hop IPv4/IPv6 address */
   NPF_MPLS_LabelStack_t labelStack;      /* label stack to be pushed*/
   NPF_MPLS_DSCP_EXP_t   *dscpToExp;      /* DSCP to EXP map */
} NPF_MPLS_NHLFE_Param_t;

typedef enum {
   NPF_MPLS_NHLFE_HANDLE = 1,
   NPF_MPLS_NHLFE_PARAMS = 2
} NPF_MPLS_NHLFE_InfoType_t;

typedef struct {
   NPF_MPLS_NHLFE_InfoType_t  paramType;   /* Handle or NHLFE information */
   union {
      NPF_MPLS_NHLFE_Handle_t nhlfeHandle; /* NHLFE Handle */
      NPF_MPLS_NHLFE_Param_t  nhlfeParam;  /* NHLFE Parameters */
     } u;
} NPF_MPLS_NHLFE_t;

typedef enum {
   NPF_MPLS_POLICYTYPE_NONE   = 0,
   NPF_MPLS_POLICYTYPE_WEIGHT = 1,
   NPF_MPLS_POLICYTYPE_ELSP   = 2,
   NPF_MPLS_POLICYTYPE_LLSP   = 3
} NPF_MPLS_NHLFE_SET_PolicyType_t;

typedef struct{
   NPF_uint32_t weight;
} NPF_MPLS_WeightPolicy_t;


typedef struct{
   NPF_uint8_t       dscp; /* incoming DSCP to select on */
} NPF_MPLS_DS_Policy_t;

typedef struct{
   NPF_MPLS_NHLFE_t *nhlfe;
   union {
      NPF_MPLS_WeightPolicy_t weightPolicy;
      NPF_MPLS_DS_Policy_t    dsPolicy;
   } u;
} NPF_MPLS_Policy_t;

typedef struct {
   NPF_uint32_t                    setId;
   NPF_MPLS_NHLFE_SET_PolicyType_t policyType;
   NPF_uint32_t                    numPolicy;
   NPF_MPLS_Policy_t               **policyArray;
} NPF_MPLS_NHLFE_SET_Param_t;

typedef NPF_uint32_t NPF_MPLS_NHLFE_SET_Handle_t;

typedef enum {
   NPF_MPLS_NHLFESET_HANDLE = 1,
   NPF_MPLS_NHLFESET_PARAMS = 2
} NPF_MPLS_NHLFE_SET_Type_t;

typedef struct {
   NPF_MPLS_NHLFE_SET_Type_t setType;  /* Handle or NHLFE Set information */
   union {
      NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;/* NHLFE Set Handle      */
      NPF_MPLS_NHLFE_SET_Param_t  nhlfeSetParam; /* NHLFE Set Parameters  */
    } u;
} NPF_MPLS_NHLFE_SET_t;

typedef struct{
   union {
      NPF_IPv4Prefix_t ipv4DestNetPrefix;   /* IPv4 prefix       */
      NPF_IPv6Prefix_t ipv6DestNetPrefix;   /* IPv6 prefix       */
      NPF_IPv4Address_t ipv4DestHostAddr;   /* IPv4 Host Address */
      NPF_IPv6Address_t ipv6DestHostAddr;   /* IPv6 Host Address */
   } u;
} NPF_MPLS_FEC_Param_t;

typedef enum {
   NPF_MPLS_FEC_IPV4_DEST_PREFIX = 1, /* IPv4 prefix       */
   NPF_MPLS_FEC_IPV4_HOSTADDR    = 2, /* IPv4 Host Address */
   NPF_MPLS_FEC_IPV6_DEST_PREFIX = 3, /* IPv6 prefix       */
   NPF_MPLS_FEC_IPV6_HOSTADDR    = 4  /* IPv6 Host Address */
} NPF_MPLS_FEC_Type_t;

typedef struct{
   NPF_MPLS_FEC_Type_t  fecType;
   NPF_MPLS_FEC_Param_t param;
} NPF_MPLS_FEC_t;

typedef enum {
   NPF_MPLS_REDIRECT            = 1,
   NPF_MPLS_COPY_PROCESS_OPCODE = 2
```

```
} NPF_MPLS_Modifier_t;


typedef enum {
   NPF_MPLS_POP_AND_LOOKUP     = 1,
   NPF_MPLS_POP_AND_FORWARD    = 2,
   NPF_MPLS_NO_POP_AND_FORWARD = 3,
   NPF_MPLS_DISCARD            = 4
} NPF_MPLS_LabelAction_t;


typedef struct {
   NPF_MPLS_Label_t       incomingLabel;    /* Incoming label*/
   NPF_IfHandle_t         ingressInterface; /* Incoming interface */
   NPF_MPLS_LabelAction_t labelAction;      /* Label action */
   NPF_MPLS_Modifier_t    lspModifyType;    /* Additional processing during
                                               the handling of packet */
   NPF_MPLS_DSCP_EXP_t    *expToDscp;       /* EXP to DSCP table associated
                                               with ELSP*/
} NPF_MPLS_ILM_t;


typedef enum{
   NPF_MPLS_LSP_FEC = 1, /*Associates FEC with NHLFE */
   NPF_MPLS_LSP_ILM = 2  /*Associates ILM with NHLFE */
   NPF_MPLS_LSP_TUN = 3  /*Creates a tunnel endpoint */
} NPF_MPLS_LSP_Type_t;


typedef NPF_uint32_t NPF_MPLS_LSP_Id_t;


typedef struct {
   NPF_MPLS_LSP_Type_t     lspType;     /* Type of LSP*/
   NPF_MPLS_LSP_Id_t       lspId;       /*LSP Tunnel Parameter - Identifier*/
   NPF_MPLS_TE_Param_t     *teParams;   /*tunnel/LSP parameters*/
   NPF_uint16_t            lspMtu;      /*LSP MTU */
   union {
      NPF_MPLS_FEC_t       *fec;        /* FEC */
      NPF_MPLS_ILM_t       *ilm;        /* ILM */
   } u;
   NPF_MPLS_DS_LSP_Model_t dsModel;      /*pipe, short pipe or uniform  */
   NPF_MPLS_DS_LSP_Type_t  dsLspType;    /*E-LSP, L-LSP, none           */
    NPF_uint16_t          ttlDecrement;  /*let SAPI or below figure out
                                          where to decrement            */
   NPF_MPLS_NHLFE_SET_t    *nhlfeSet;    /* Associated NHLFE Set        */
} NPF_MPLS_LSP_t;

typedef NPF_uint32_t NPF_MPLS_LSP_Handle_t;

typedef enum {
   NPF_MPLS_LSP_HANDLE = 1,
   NPF_MPLS_LSP_PARAMS = 2
} NPF_MPLS_LSP_InfoType_t;


typedef struct {
   NPF_MPLS_LSP_InfoType_t  paramType; /* Handle or LSP information */
   union {
      NPF_MPLS_LSP_Handle_t lspHandle; /* LSP Handle */
      NPF_MPLS_LSP_t        lspParam;  /* LSP key values */
   } u;
} NPF_MPLS_LSP_Info_t;

typedef struct {
   NPF_uint64_t octetsRcvd;  /* Total Rx Octets */
   NPF_uint64_t packetsRcvd; /* Total Rx Packets */
   NPF_uint64_t errors;      /* Erroneous packets discarded */
   NPF_uint64_t drops;       /* Non erroneous packets discarded */
}NPF_MPLS_FEC_Stats_t;

typedef struct {
   NPF_uint64_t octetsRcvd;  /* Total Rx Octets                 */
   NPF_uint64_t packetsRcvd; /* Total Rx Packets                */
   NPF_uint64_t errors;      /* Erroneous packets discarded     */
   NPF_uint64_t drops;       /* Non erroneous packets discarded */
} NPF_MPLS_ILM_Stats_t;

typedef struct {
   NPF_uint64_t bytes;   /* byte count   */
   NPF_uint64_t packets; /* packet count */
} NPF_MPLS_DSCP_EXP_EntryStat_t;
typedef struct {
   NPF_uint8_t numEntries;
   NPF_MPLS_DSCP_EXP_EntryStat_t *stats; /*stats associated with entries*/
} NPF_MPLS_DSCP_EXP_Stats_t;

typedef struct {
   NPF_MPLS_NHLFE_Handle_t nhlfeHandle; /* NHLFE Handle                    */
   NPF_uint64_t            octetsTxed;  /* Total Tx Octets                 */
   NPF_uint64_t            packetsTxed; /* Total Tx Packets                */
   NPF_uint64_t            errors;      /* Erroneous packets discarded     */
   NPF_uint64_t            drops;       /* Non erroneous packets discarded */
} NPF_MPLS_NHLFE_Stats_t;

typedef struct {
   NPF_uint32_t           nhlfeCount;
   NPF_MPLS_NHLFE_Stats_t *nhlfeStatsArray;
}NPF_MPLS_NHLFE_StatsArray_t;

typedef struct {
    NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;   /* NHLFE Set Handle */
    NPF_MPLS_NHLFE_StatsArray_t *nhlfeStatsArray  /* Array of NHLFE
                                                          Statistics    */
} NPF_MPLS_NHLFE_SET_Stats_t;

typedef struct {
   NPF_uint32_t               nhlfeSetCount;
   NPF_MPLS_NHLFE_SET_Stats_t *nhlfeSetStatsArray;
} NPF_MPLS_NHLFE_SET_StatsArray_t;

typedef struct {
   NPF_MPLS_LSP_Type_t     lspType;       /* Type of LSP    */
   NPF_MPLS_LSP_Handle_t   lspHandle;     /* LSP identifier */
   union {
      NPF_MPLS_FEC_Stats_t fecStatistics; /* FEC */
      NPF_MPLS_ILM_Stats_t ilmStatistics; /* ILM */
   } u;
   NPF_MPLS_NHLFE_StatsArray_t nhlfeStatsArray; /* Associated NHLFEs*/
} NPF_MPLS_LSP_Stats_t;


typedef enum NPF_MPLSCallbackType {
   NPF_MPLS_LSP_ENTRY_CREATE          = 1,
   NPF_MPLS_LSP_ENTRY_DELETE          = 2,
   NPF_MPLS_LSP_ENTRY_MODIFY          = 3,
   NPF_MPLS_LSP_ATTRIBUTE_QUERY       = 4,
   NPF_MPLS_LSP_ENTRY_QUERY           = 5,
   NPF_MPLS_LSP_STATS_QUERY           = 6,
   NPF_MPLS_NHLFE_ENTRY_CREATE        = 7,
   NPF_MPLS_NHLFE_ENTRY_DELETE        = 8,
   NPF_MPLS_NHLFE_ENTRY_MODIFY        = 9,
   NPF_MPLS_NHLFE_ATTRIBUTE_QUERY     = 10,
   NPF_MPLS_NHLFE_ENTRY_QUERY         = 11,
   NPF_MPLS_NHLFE_STATS_QUERY         = 12,
   NPF_MPLS_NHLFE_SET_CREATE          = 13,
   NPF_MPLS_NHLFE_SET_DELETE          = 14,
   NPF_MPLS_NHLFE_SET_MODIFY          = 15,
   NPF_MPLS_NHLFE_SET_ATTRIBUTE_QUERY = 16,
   NPF_MPLS_NHLFE_SET_ENTRY_QUERY     = 17,
   NPF_MPLS_NHLFE_SET_STATS_QUERY     = 18,
   NPF_MPLS_DSCPEXP_ENTRY_CREATE      = 19,
   NPF_MPLS_DSCPEXP_ENTRY_DELETE      = 20,
   NPF_MPLS_DSCPEXP_ENTRY_MODIFY      = 21,
   NPF_MPLS_DSCPEXP_ATTRIBUTE_QUERY   = 22,
   NPF_MPLS_DSCPEXP_ENTRY_QUERY       = 23,
   NPF_MPLS_DSCPEXP_STATS_QUERY       = 24
} NPF_MPLS_CallbackType_t;


typedef struct {
   NPF_uint32_t                    lspArrayIndex;
   NPF_MPLS_ReturnCode_t           returnCode;
   NPF_MPLS_LSP_Handle_t           lspHandle;
   NPF_MPLS_NHLFE_SET_CreateResp_t nhlfeSetResp;
   NPF_MPLS_DSCP_EXP_Handle_t      expDscpHandle;
} NPF_MPLS_LSP_CreateResp_t;


typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_LSP_Handle_t      lspHandle;
   NPF_MPLS_LSP_t             lspEntry;
   NPF_MPLS_DSCP_EXP_Handle_t expDscpHandle;
} NPF_MPLS_LSP_EntryResp_t;


typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_DSCP_EXP_Handle_t expDscpHandle;
} NPF_MPLS_DSCP_EXP_CreateResp_t;

typedef struct {
    NPF_uint32_t               arrayIndex;
    NPF_MPLS_ReturnCode_t      returnCode;
    NPF_MPLS_DSCP_EXP_Handle_t expDscpHandle;
    NPF_MPLS_DSCP_EXP_Type_t   type;
    NPF_MPLS_DSCP_EXP_Param_t  dscpExpEntry;
} NPF_MPLS_DSCP_EXP_EntryResp_t;

typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_NHFLE_Handle_t    nhlfeHandle;
   NPF_MPLS_DSCP_EXP_Handle_t dscpExpHandle;
} NPF_MPLS_NHFLE_CreateResp_t;

typedef struct {
   NPF_uint32_t               arrayIndex;
   NPF_MPLS_ReturnCode_t      returnCode;
   NPF_MPLS_NHLFE_Handle_t    nhlfeHandle;
   NPF_MPLS_NHLFE_Param_t     nhlfeEntry;
   NPF_MPLS_DSCP_EXP_Handle_t dscpExpHandle;
} NPF_MPLS_NHLFE_EntryResp_t;

typedef struct {
   NPF_uint32_t                arrayIndex;
   NPF_MPLS_ReturnCode_t       returnCode;
   NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;
   NPF_uint32_t                numNhlfeResp;
   NPF_MPLS_NHLFE_CreateResp_t **numNhlfeResp;
} NPF_MPLS_NHLFE_SET_CreateResp_t;

typedef struct {
   NPF_uint32_t                arrayIndex;
   NPF_MPLS_ReturnCode_t       returnCode;
   NPF_MPLS_NHLFE_SET_Handle_t nhlfeSetHandle;
   NPF_MPLS_NHLFE_SET_Param_t  nhlfeEntry;
} NPF_MPLS_NHLFE_SET_EntryResp_t;

typedef struct {
   NPF_MPLS_ReturnCode_t returnCode; /* Return code for the call */
   union {
      NPF_MPLS_LSP_CreateResp_t       lspCreateResp;
      NPF_MPLS_LSP_EntryResp_t        lspEntryResp;
      NPF_MPLS_LSP_Handle_t           lspHandle;
      NPF_MPLS_LSP_Stats_t            lspStatsResp;
      NPF_MPLS_NHLFE_CreateResp_t     nhlfeCreateResp;
      NPF_MPLS_NHLFE_EntryResp_t      nhlfeEntryResp;
      NPF_MPLS_NHLFE_Handle_t         nhlfeHandle;
      NPF_MPLS_NHLFE_Stats_t          nhlfeStatsResp;
      NPF_MPLS_NHLFE_SET_Handle_t     nhlfeSetHandle;
      NPF_MPLS_NHLFE_SET_CreateResp_t nhlfeSetCreateResp;
      NPF_MPLS_NHLFE_SET_EntryResp_t  nhlfeSetEntryResp;
      NPF_MPLS_NHLFE_SET_Stats_t      nhlfeSetStatsResp;
      NPF_MPLS_DSCP_EXP_CreateResp_t  dscpExpCreateResp;
      NPF_MPLS_DSCP_EXP_EntryResp_t   dscpExpEntryResp;
      NPF_MPLS_DSCP_EXP_Stats_t       dscpExpStats;
      NPF_MPLS_DSCP_EXP_Handle_t      dscpExpHandle;
      NPF_uint32_t                    tableSpaceRemaining;
      NPF_uint32_t                    unused;
   } u;
} NPF_MPLS_AsyncResponse_t;

typedef struct {
   NPF_MPLS_CallbackType_t  type;    /* Callback function type            */
   NPF_boolean_t            allOk;   /* TRUE if all functions completed OK */
   NPF_uint32_t             numResp; /* Number of responses in array      */
   NPF_MPLS_AsyncResponse_t *resp;   /* Pointer to response structures     */
} NPF_MPLS_CallbackData_t;

#define NPF_MPLS_E_ALREADY_REGISTERED           (NPF_MPLS_BASE_ERR+1)
#define NPF_MPLS_E_BAD_CALLBACK_HANDLE          (NPF_MPLS_BASE_ERR+2)
#define NPF_MPLS_E_BAD_CALLBACK_FUNCTION        (NPF_MPLS_BASE_ERR+3)
#define NPF_MPLS_E_INVALID_LSP_PARAM            (NPF_MPLS_BASE_ERR+4)
#define NPF_MPLS_E_INVALID_LSP_HANDLE           (NPF_MPLS_BASE_ERR+5)
#define NPF_MPLS_E_INVALID_LSP_TYPE             (NPF_MPLS_BASE_ERR+6)
#define NPF_MPLS_E_INVALID_NHLFE_PARAM          (NPF_MPLS_BASE_ERR+7)
#define NPF_MPLS_E_INVALID_NHLFE_HANDLE         (NPF_MPLS_BASE_ERR+8)
#define NPF_MPLS_E_INVALID_NHLFESET_PARAM       (NPF_MPLS_BASE_ERR+9)
#define NPF_MPLS_E_INVALID_NHLFESET_HANDLE      (NPF_MPLS_BASE_ERR+10)
#define NPF_MPLS_E_INVALID_FEC                  (NPF_MPLS_BASE_ERR+11)
#define NPF_MPLS_E_INVALID_IN_LABEL             (NPF_MPLS_BASE_ERR+12)
#define NPF_MPLS_E_INVALID_OUT_LABEL            (NPF_MPLS_BASE_ERR+13)
#define NPF_MPLS_E_INVALID_LABEL_STACK          (NPF_MPLS_BASE_ERR+14)
#define NPF_MPLS_E_INVALID_NEXT_HOP_IP          (NPF_MPLS_BASE_ERR+15)
#define NPF_MPLS_E_INVALID_NEXT_HOP_L2MEDIA     (NPF_MPLS_BASE_ERR+16)
#define NPF_MPLS_E_INVALID_NHLFE_FWD_POLICY     (NPF_MPLS_BASE_ERR+17)
#define NPF_MPLS_E_INVALID_INTERFACE            (NPF_MPLS_BASE_ERR+18)
#define NPF_MPLS_E_UNKNOWN                      (NPF_MPLS_BASE_ERR+19)
#define NPF_MPLS_E_INVALID_DSCPEXP_HANDLE       (NPF_MPLS_BASE_ERR+20)
#define NPF_MPLS_E_ENTRY_ALREADY_EXIST          (NPF_MPLS_BASE_ERR+21)
#define NPF_MPLS_E_INSUFFICIENT_STORAGE         (NPF_MPLS_BASE_ERR+22)
#define NPF_MPLS_E_FUNCTION_NOT_SUPPORTED       (NPF_MPLS_BASE_ERR+23)


typedef enum {
   NPF_MPLS_EV_ILM_MISS  = 1,   /* No ILM entry for label */
   NPF_MPLS_EV_ILM_NHLFE = 2,   /* Packet matches ILM without NHLFE */
   NPF_MPLS_EV_FTN_NHLFE = 3,   /* Packet matches FTN without NHLFE */
   NPF_MPLS_EV_NHLFE_MTU = 4,   /* Labeled packet exceeds MTU */
   NPF_MPLS_EV_NHLFE_L2  = 5,   /* Need next hop resolution */
   NPF_MPLS_EV_PKT_TTL   = 6,   /* Exceeded TTL */
   NPF_MPLS_EV_NHLFE_MISS_EVENT = 7 /* When NHLFE/NHLFE SET does not exist */
} NPF_MPLS_EventType_t;


typedef struct {
   NPF_MPLS_Event_t eventType;  /* Event type */
   union {
      NPF_MPLS_LSP_Handle_t   lspHandle;
      NPF_MPLS_NHLFE_Handle_t nhlfeHandle;
   } u;
   NPF_IfHandle_t ingressInterface;
   NPF_uint32_t   packetLength; /* Length of packet   */
   void           *packetData;  /* Location of packet */
} NPF_MPLS_EventData_t;


typedef struct    {
   NPF_uint16_t         nData;      /* Number of events in array    */
   NPF_MPLS_EventData_t *eventData; /* Array of event notifications */
} NPF_MPLS_EventArray_t;

/* MPLS SAPI Function Prototypes */

   typedef void (*NPF_MPLS_CallbackFunc_t) (
            NPF_IN NPF_userContext_t userContext,
            NPF_IN NPF_correlator_t correlator,
            NPF_IN NPF_MPLS_CallbackData_t *callbackData);

   typedef void (*NPF_MPLS_EventCallFunc_t) (
            NPF_IN NPF_userContext_t     userContext,
            NPF_IN NPF_MPLS_EventArray_t mplsEventArray);


   NPF_error_t NPF_MPLS_Register(
            NPF_IN NPF_userContext_t         userContext,
            NPF_IN NPF_MPLS_FwCallbackFunc_t callbackFunc,
            NPF_OUT NPF_callbackHandle_t     *callbackHandle);


   NPF_error_t NPF_MPLS_Deregister(

            NPF_IN NPF_callbackHandle_t callbackHandle);


   NPF_error_t NPF_MPLS_EventRegister(
            NPF_IN  NPF_userContext_t             userContext,
            NPF_IN  NPF_MPLS_EventHandlerFunc_t   eventCallFunc,
            NPF_OUT NPF_callbackHandle_t          *eventCallHandle);


   NPF_error_t NPF_MPLS_EventDeregister(
            NPF_IN NPF_callbackHandle_t eventCallHandle);


   NPF_error_t  NPF_MPLS_LSP_EntryCreate(
            NPF_IN NPF_callbackHandle_t  callbackHandle,
            NPF_IN NPF_correlator_t      correlator,
            NPF_IN NPF_errorReporting_t  errorReporting,
            NPF_IN NPF_uint32_t          nMplsLsp,
            NPF_IN NPF_MPLS_LSP_t        **mplsLspArray);


   NPF_error_t  NPF_MPLS_LSP_EntryDelete(
            NPF_IN NPF_callbackHandle_t  callbackHandle,
            NPF_IN NPF_correlator_t      correlator,
            NPF_IN NPF_errorReporting_t  errorReporting,
            NPF_IN NPF_uint32_t          nHandles,
            NPF_IN NPF_MPLS_LSP_Handle_t *mplsLspHandleArray);


   NPF_error_t  NPF_MPLS_LSP_EntryModify(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting,
            NPF_IN NPF_uint32_t           nHandles,
            NPF_IN NPF_MPLS_LSP_Handle_t  *mplsLspHandleArray,
            NPF_IN NPF_MPLS_LSP_t         *mplsLspArray);

   NPF_error_t  NPF_MPLS_LSP_EntryQuery(
            NPF_IN NPF_callbackHandle_t callbackHandle,
            NPF_IN NPF_correlator_t     correlator,
            NPF_IN NPF_errorReporting_t errorReporting,
            NPF_IN NPF_uint32_t         numInfo,
            NPF_IN NPF_MPLS_LSP_Info_t  *mplsLspInfoArray);

   NPF_error_t  NPF_MPLS_LSP_StatsGet(
            NPF_IN NPF_callbackHandle_t  callbackHandle,
            NPF_IN NPF_correlator_t      correlator,
            NPF_IN NPF_errorReporting_t  errorReporting,
            NPF_IN NPF_uint32_t          nHandles,
            NPF_IN NPF_MPLS_LSP_Handle_t *mplsLspHandleArray);

   NPF_error_t NPF_MPLS_LSP_AttributeQuery(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting);

   NPF_error_t NPF_MPLS_DSCP_EXP_Create(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting,
            NPF_IN NPF_MPLS_DSCP_EXP_Type_t type,
            NPF_IN NPF_uint32_t nTables,
            NPF_IN NPF_MPLS_DSCP_EXP_Param_t **dscpExptablesArray);

   NPF_error_t  NPF_MPLS_DSCP_EXP_Delete(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting,
            NPF_IN NPF_uint32_t           nHandles,
            NPF_IN NPF_MPLS_DSCP_EXP_Handle_t *dscpExpHandleArray);

   NPF_error_t  NPF_MPLS_DSCP_EXP_Modify(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting,
            NPF_IN NPF_uint32_t nHandles,
            NPF_IN NPF_MPLS_DSCP_EXP_Handle_t *dscpExpHandleArray,
            NPF_IN NPF_MPLS_DSCP_EXP_Param_t  *dscpExpArray);

   NPF_error_t  NPF_MPLS_DSCP_EXP_StatsGet(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting,
            NPF_IN NPF_uint32_t nHandles,
            NPF_IN NPF_MPLS_DSCP_EXP_Handle_t *dscpExpHandleArray);

   NPF_error_t NPF_MPLS_DSCP_EXP_AttributeQuery(
            NPF_IN NPF_callbackHandle_t   callbackHandle,
            NPF_IN NPF_correlator_t       correlator,
            NPF_IN NPF_errorReporting_t   errorReporting);
```

```
NPF_error_t  NPF_MPLS_DSCP_EXP_EntryQuery(
         NPF_IN NPF_callbackHandle_t callbackHandle,
         NPF_IN NPF_correlator_t     correlator,
         NPF_IN NPF_errorReporting_t errorReporting,
         NPF_IN NPF_uint32_t         numTables,
         NPF_IN NPF_MplsDscpExp_t    *dscpExpArray);


NPF_error_t NPF_MPLS_NHLFE_Create(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          nNhlfe,
         NPF_IN NPF_MPLS_NHLFE_t      **mplsNhlfeArray);


NPF_error_t  NPF_MPLS_NHLFE_Delete(
         NPF_IN NPF_callbackHandle_t    callbackHandle,
         NPF_IN NPF_correlator_t        correlator,
         NPF_IN NPF_errorReporting_t    errorReporting,
         NPF_IN NPF_uint32_t            nHandles,
         NPF_IN NPF_MPLS_NHLFE_Handle_t *mplsNhlfeHandleArray);


NPF_error_t  NPF_MPLS_NHLFE_Modify(
         NPF_IN NPF_callbackHandle_t    callbackHandle,
         NPF_IN NPF_correlator_t        correlator,
         NPF_IN NPF_errorReporting_t    errorReporting,
         NPF_IN NPF_uint32_t            nHandles,
         NPF_IN NPF_MPLS_NHLFE_Handle_t *mplsNhlfeHandleArray,
         NPF_IN NPF_MPLS_NHLFE_t        *mplsNhlfeArray);


NPF_error_t  NPF_MPLS_NHLFE_StatsGet(
         NPF_IN NPF_callbackHandle_t    callbackHandle,
         NPF_IN NPF_correlator_t        correlator,
         NPF_IN NPF_errorReporting_t    errorReporting,
         NPF_IN NPF_uint32_t            nHandles,
         NPF_IN NPF_MPLS_NHLFE_Handle_t *mplsNhlfeHandleArray);

NPF_error_t NPF_MPLS_NHLFE_AttributeQuery(
         NPF_IN NPF_callbackHandle_t   callbackHandle,
         NPF_IN NPF_correlator_t       correlator,
         NPF_IN NPF_errorReporting_t   errorReporting);


NPF_error_t  NPF_MPLS_NHLFE_EntryQuery(
         NPF_IN NPF_callbackHandle_t callbackHandle,
         NPF_IN NPF_correlator_t     correlator,
         NPF_IN NPF_errorReporting_t errorReporting,
         NPF_IN NPF_uint32_t         numNhlfe,
         NPF_IN NPF_MPLS_NHLFE_t     *mplsNhlfeArray);


NPF_error_t  NPF_MPLS_NHLFE_SET_Create(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          nNhlfeSet,
         NPF_IN NPF_MPLS_NHLFE_SET_t  **mplsNhlfeSetArray);


NPF_error_t  NPF_MPLS_NHLFE_SET_Delete(
         NPF_IN NPF_callbackHandle_t        callbackHandle,
         NPF_IN NPF_correlator_t            correlator,
         NPF_IN NPF_errorReporting_t        errorReporting,
         NPF_IN NPF_uint32_t                nHandles,
         NPF_IN NPF_MPLS_NHLFE_SET_Handle_t *mplsNhlfeSetHandleArray);

NPF_error_t  NPF_MPLS_NHLFE_SET_Modify(
         NPF_IN NPF_callbackHandle_t        callbackHandle,
         NPF_IN NPF_correlator_t            correlator,
         NPF_IN NPF_errorReporting_t        errorReporting,
         NPF_IN NPF_uint32_t                nNhlfe,
         NPF_IN NPF_MPLS_NHLFE_SET_Handle_t *mplsNhlfeSetHandleArray,
         NPF_IN NPF_MPLS_NHLFE_SET_t        *mplsNhlfeSetArray);

NPF_error_t  NPF_MPLS_NHLFE_SET_EntryQuery(
         NPF_IN NPF_callbackHandle_t  callbackHandle,
         NPF_IN NPF_correlator_t      correlator,
         NPF_IN NPF_errorReporting_t  errorReporting,
         NPF_IN NPF_uint32_t          numNhlfeSet,
         NPF_IN NPF_MPLS_NHLFE_SET_t  *nhlfeSetArray);

NPF_error_t  NPF_MPLS_NHLFE_StatsGet(
         NPF_IN NPF_callbackHandle_t        callbackHandle,
         NPF_IN NPF_correlator_t            correlator,
         NPF_IN NPF_errorReporting_t        errorReporting,
         NPF_IN NPF_uint32_t                nHandles,
         NPF_IN NPF_MPLS_NHLFE_SET_Handle_t *mplsNhlfeSetHandleArray);

NPF_error_t NPF_MPLS_NHLFE_SET_AttributeQuery(
         NPF_IN NPF_callbackHandle_t   callbackHandle,
         NPF_IN NPF_correlator_t       correlator,
         NPF_IN NPF_errorReporting_t   errorReporting);
```

Appendix B MPLS QoS Parameters

```
typedef struct {
      NPF_uint16_t     flag;                 /*specifies valid params*/
      NPF_uint32_t     maxRate;              /*Max or Peak rate (bps)*/
      NPF_uint32_t     meanRate;             /*Mean rate (bps)*/
      NPF_uint32_t     maxBurstSize;         /*Max Burst size in bytes*/
      NPF_uint32_t     meanBurstSize;        /*Mean Burst size in bytes*/
      NPF_uint32_t     exBurstSize;          /*Excess Burst size in bytes*/
      NPF_uint32_t     frequency;            /*Frequency of token refresh*/
      NPF_uint8_t      weight;               /*Weight associated with tunnel*/
      NPF_uint8_t      trafficClass;         /*Derived from parameters above*/
} NPF_MPLS_TE_Param_t;

maxRate = PDR
meanRate = CBR
maxBurst = PBS
meanBurst = CBS
exBurst = EBS
frequency = service frequency
weight = weight
trafficClass: derived from table 1.
```

The following chart is taken from [RFC 3212] and gives examples of services and what the corresponding CR-LDP parameters would be.
The key parameter to providing QoS, as opposed to bandwidth management, is the Service Frequency. This has the most effect on queuing delay and delay variation. As can be seen from the table, Unspecified is enough to provide most services. Real-time traffic would require support of Frequent. Very Frequent need only be supported, to provide circuit emulation (e.g., DS1 and DS3 service) and real-time layer 2 transport services (e.g., ATM CBR, VBR nrt, low delay Frame Relay services). QoS is necessary to support [draft-martini-l2circuit-trans-mpls-06], and perhaps some Diffserv classes.
NOTE: Unspecified frequency is adequate. Phase II allows all but Very Frequent.

Service Example	**PDR**	**PBS**	**CDR**	**CBS**	**EBS**	**Service Freq**	**Conditioning Action**	**Diffserv Class**
Delay Sensitive (DS)	S	S	=PDR	=PBS	0	Frequent	drop > PDR	EF
Throughput Sensitive (TS)	S	S	S	S	0	Unspecified	drop > PDR, PBS; mark > CDR, CBS	AF
Best Effort (BE)	Inf	inf	inf	inf	0	Unspecified	-	BE
Frame Relay Service	S	S	CIR	Bc	Be	Unspecified	drop > PDR,PBS; mark > CDR,CBS,EBS	AF
ATM-CBR	PCR	CDVT	=PCR	=CDVT	0	VeryFrequent	drop > PCR	N/A

Service Example	**PDR**	**PBS**	**CDR**	**CBS**	**EBS**	**Service Freq**	**Conditioning Action**	**Diffserv Class**
ATM-VBR.3 (rt)	PCR	CDVT	SCR	MBS	0	Frequent	drop > PCR; mark > SCR, MBS	EF
ATM-VBR.3 (nrt)	PCR	CDVT	SCR	MBS	0	Unspecified	drop > PCR; mark > SCR, MBS	AF
ATM-UBR	PCR	CDVT	-(0)	-(0)	0	Unspecified	drop > PCR	BE
ATM-GFR.1	PCR	CDVT	MCR	MBS	0	Unspecified	drop > PCR	AF
ATM-GFR.2	PCR	CDVT	MCR	MBS	0	Unspecified	drop > PCR; mark >MCR,MFS	AF / higher drop precedence
int-serv-Control Load (CL)	P	m	r	b	0	Frequent	drop > p; drop >r,b	EF

NOTES:

S	User Specified
inf	interface
CIR	Committed Information Rate
Bc	Committed Burst Size
Be	Excess Burst Size
PDR	Peak Data Rate
PBS	Peak Burst Size
PCR	Peak Cell Rate
SCR	Sustainable Cell Rate
MBS	Maximum Burst Size
CDVT	Cell Delay Variation Tolerance
CDR	Committed Data Rate
CBS	Committed Burst Size
EBS	Excess Burst Size
p	peak rate of the CL (Controlled Load) service
m	min. packet size
r	data rate of the CL service
b	burst

Table 1 "CRLDP Parameter Mapping to Traffic Class" below covers most of the cases from CRLDP.

CRLDP Parameters	**ATM Type Traffic Class**
PDR, PBS > 0; CDR, CBS = PDR, PBS; frequency=veryfrequent	CBR
PDR, PBS > 0; PDR, PBS >= CDR, CBS > 0; EBS >= 0; frequency=frequent;	VBR.3-

weight >=0	RT
PDR, PBS > 0; PDR, PBS >= CDR, CBS > 0; EBS >= 0; frequency=unspecified; weight >= 0	VBR.3-NRT
PDR, PBS > 0; CDR, CBS = 0; EBS >= 0; frequency=unspecified; weight >= 0	UBR

Table 2 - CRLDP Parameter Mapping to Traffic Class

Appendix C Relationship of MPLS SAPI with IPv4/IPv6 SAPI for FTN Mapping

The MPLS SAPI relies on the function of the IPv4 and IPv6 route entries to construct an FTN mapping. The FEC is the route entry prefix. It is assumed that the MPLS application has informed the routing application of an NHLFE Set that can be used as a next hop for a given prefix, i.e., FEC. The IPv4 or IPv6 SAPI is used to install a NHLFE Set as a possible next hop for the prefix. The route entry reflecting the prefix may point to an NHLFE Set as the active next hop. If a received unlabeled packet matches the route entry, and the active next hop is an NHFLE Set, the unlabeled packet is associated with that NHLFE Set. The system then forwards the packet to the egress interface where the packet is labeled from the selected NHLFE and transmitted.

Appendix D List of companies belonging to NPF during approval process

Agere Systems

Alcatel

Altera

AMCC

Analog Devices

Avici Systems

Azanda Network Devices

Cypress Semiconductor

Ericsson

Erlang Technologies

EZ Chip

Flextronics

Fujitsu Ltd.

FutureSoft

HCL Technologies

Hi/fn

IBM

IDT

Intel

IP Infusion

Kawasaki LSI

LSI Logic

Modelware

Mosaid

Motorola

NEC

NetLogic

Nokia

Paion Co., Ltd.

PMC Sierra

RadiSys

Samsung Electronics

Sandburst Corporation

Silicon & Software Systems

Silicon Access

Sony Electronics

STMicroelectronics

Sun Microsystems

Teja Technologies

TranSwitch

U4EA Group

Xelerated

Xilinx

Zettacom

ZTE