Messaging LFB
Implementation Agreement

November 2, 2004
Revision 1.1

Editor(s):
Gamil Cain, Intel Corporation, gamil.cain@intel.com

Copyright © 2004 The Network Processing Forum (NPF). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction other than the following, (1) the above copyright notice and this paragraph must be included on all such copies and derivative works, and (2) this document itself may not be modified in any way, such as by removing the copyright notice or references to the NPF, except as needed for the purpose of developing NPF Implementation Agreements.

By downloading, copying, or using this document in any manner, the user consents to the terms and conditions of this notice. Unless the terms and conditions of this notice are breached by the user, the limited permissions granted above are perpetual and will not be revoked by the NPF or its successors or assigns.

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "AS IS" BASIS WITHOUT ANY WARRANTY OF ANY KIND. THE INFORMATION, CONCLUSIONS AND OPINIONS CONTAINED IN THE DOCUMENT ARE THOSE OF THE AUTHORS, AND NOT THOSE OF NPF. THE NPF DOES NOT WARRANT THE INFORMATION IN THIS DOCUMENT IS ACCURATE OR CORRECT. THE NPF DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED THE IMPLIED LIMITED WARRANTIES OF MERCHANTABILITY, TITLE OR FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS.

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 1.0.

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

1 Revision History ........................................................................................................... 3
2 Introduction ...................................................................................................................... 4
   2.1 Acronyms / Definitions ............................................................................................. 4
   2.2 Assumptions .............................................................................................................. 5
   2.3 Scope ......................................................................................................................... 5
   2.4 External Requirements and Dependencies ............................................................... 5
3 LFB Detailed Description .............................................................................................. 6
   3.1 LFB Input Ports .......................................................................................................... 8
   3.2 LFB Output Ports ........................................................................................................ 8
   3.3 Relationship with other LFBs .................................................................................... 9
4 Data Types .................................................................................................................. 11
   4.1 Common LFB Data Types ......................................................................................... 11
   4.2 Data Structures for Completion Callbacks ............................................................... 12
   4.3 Data Structures for Event Notifications ................................................................... 13
   4.4 Error Codes ............................................................................................................... 13
5 Functional APIs (FAPIs) ............................................................................................. 14
   5.1 Required APIs .......................................................................................................... 14
   5.2 Optional APIs .......................................................................................................... 14
6 References .................................................................................................................. 23

Appendix A Npf_f_msg.h .................................................................................................... 24
Appendix B List of companies belonging to NPF DURING APPROVAL PROCESS and
Acknowledgements .......................................................................................................... 29

Table of Figures

Figure 1 -- Conceptual model of Messaging LFB ................................................................. 4
Figure 2 -- Two Messaging LFB examples ....................................................................... 7
Figure 3 -- The Messaging LFB diagram ......................................................................... 8
Figure 4 -- Mapping of Callback Type to Callback Data ................................................. 12
1 Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Reason for Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>04/14/2004</td>
<td>Created Rev 1.0 of the implementation agreement by taking the Messaging LFB draft (npf2003.504.07) and making minor editorial corrections.</td>
</tr>
<tr>
<td>1.1</td>
<td>12/16/2004</td>
<td>Updated to add LFB type code.</td>
</tr>
</tbody>
</table>
2 Introduction

A forwarding plane, as modeled within the NPF, is comprised of a series of Logical Function Blocks (LFBs), which may span multiple forwarding elements (FEs), to perform distinct packet processing activities. Within each FE, these LFBs are connected together, via the LFB Topology [LFB_TOPO], in order to carry out a consistent forwarding plane function for the FE. In order to carry out the forwarding plane function of the FE, or in order to provide forwarding functionality across FEs, it is often necessary for these LFBs to share metadata with each other, as a downstream LFB may rely on metadata from a previous LFB as part of its processing. When the passing of packet and metadata occurs between FEs that span a physical boundary, it is necessary to have common, well-defined interfaces for accepting and receiving packet and metadata at the edges of these FEs. The mechanism by which an FE passes packet and metadata, across a physical boundary, to another FE is the Messaging LFB. The following describes the Messaging LFB and how it is used.

The function of the Messaging LFB is to pass packet and metadata between FEs, across a physical boundary (e.g. LA-1, switch fabric, etc.), in a consistently packaged manner. This allows for interoperability of FEs, from a logical perspective, even if those FEs come from different vendors. The Messaging LFB uses the NPF Messaging Layer Implementation Agreement [NPF_MSG] as the basis for how it formats the metadata. The NPF Messaging IA defines a flexible grammar for metadata syntax. Figure 1 provides a conceptual picture of how the Messaging LFB functions within the Forwarding Plane.

2.1 Acronyms / Definitions

The following terms are used in this document:

Control Plane
The control plane maintains information for the purpose of effecting changes to the forwarding plane's behavior.

Data Plane
In this document a synonym for “forwarding plane”.

Forwarding Element
An FE is modeled as a collection of logical function blocks (LFBs) whose relationship is expressed using a directed graph.

Logical Function Block
An abstraction of a defined packet-processing functionality standardized by the NPF. An NPF Functional API provides an LFB-aware means of accessing (configuring, monitoring) the resources associated with the LFB.
Messaging

Communication between processors (i.e. Control Processor and Network Processor or between Network Processors) conveying a range of configuration, dynamic state and hardware status.

Network Processor

NP

A programmable or configurable semiconductor based device that is designed and optimized for the processing of network data (packets).

Packet

The SwAPI Lexicon [7] defines the term packet in general. In this document it is used to refer to an IP packet (inclusive of header and data).

SAE

System Architectural Entity.  (Taken from [NPF_MSG]) An out-of-band configuration mechanism which gathers capabilities and requirements for the system and NPEs, and then, per-conveyance, configures the conveyance and application (i.e., service access point, or SAP) information in each NPE. Finally, the SAE determines and communicates message formats for each conveyance by mapping message fields into slots.

2.2 Assumptions

This document assumes the reader is familiar with the NPF Messaging IA [NPF_MSG] and understands how it is to be used in an NPF compliant forwarding element. Using the NPF Messaging IA terminology, the Messaging LFB exposes a control interface which allows the SAE to instantiate a specific application of the NPF Messaging specification.

2.3 Scope

The Messaging LFB describes the mechanisms by which metadata is formatted, for the purposes of transferring that metadata, along with the packet if required, between FEs that span a physical boundary. The format for metadata is based on the NPF Messaging IA [NPF_MSG]. The Messaging LFB does not define or create metadata. It only formats the metadata it receives to be compliant with the NPF Messaging IA or interprets an NPF Messaging compliant metadata stream.

2.4 External Requirements and Dependencies

The Messaging LFB describes the mechanisms by which metadata is packaged for the purposes of transferring the metadata between FEs that span a physical boundary. The packaging of metadata is based on the NPF Messaging IA [NPF_MSG]. The Messaging LFB constrains the format by which metadata is packaged to be compliant with this specification.

The FAPI Topology Discovery API [LFB_TOPO] defines the LFB ID type, NPF_BlockId_t which allows the Messaging LFB to be discovered by the Topology Discovery API.

Appendix A provides a header file which includes informative definitions of metadata elements. However, the normative definitions of metadata elements, as defined by the NPF, are contained in the LFB implementation agreements themselves. An implementation of the Messaging LFB will need to construct an appropriate header file based on NPF normative metadata definitions (i.e. the LFB implementation agreements that are to be used within an application). Appendix A is only provided as an example.
3 LFB Detailed Description

The Messaging LFB allows a system integrator to format packet and metadata (within the constraints of the underlying physical device) transferred between FEs to be compliant with the NPF Messaging IA [NPF_MSG]. The Messaging LFB, in and of itself, does not produce any metadata, nor does it perform any packet processing. The Messaging LFB is simply a formatting function that must perform two distinct operations: (1) take native packet and metadata syntax of a FE and convert it to an NPF Messaging compliant syntax and (2) translate a NPF Messaging packet / metadata stream into an FEs native syntax. Depending on the application, it may be desirable to restrict the amount of metadata that is transmitted from the Messaging LFB in order to conserve bandwidth. The Messaging LFB allows the SAE to select which metadata elements are actually placed on the “wire” by the Messaging LFB.

The Messaging LFB supports one input port and one output port on both “sides” (ingress and egress) of the Messaging LFB. This allows for an instance of the Messaging LFB to support the formatting of packet and metadata in a bi-directional manner, if so desired. Whether or not an instance of the Messaging LFB supports bi-directional formatting is up to the implementer.

The implementation of the formatting function is out of scope for the Messaging LFB, as it will vary from vendor to vendor. Implementers of the Messaging LFB will determine the most appropriate place/method to implement it. The operation of the Messaging LFB (which consists of converting native metadata syntax to NPF Messaging syntax, and vice-versa) could be expressed as a compile time or initialization option, a portion of microcode, or native to the device (e.g. an ASIC that has been hard wired to comply with the NPF Messaging specification). Additionally, the Messaging LFB provides an optional “set” API function for devices that support manipulation of metadata during runtime. The Messaging LFB also provides an optional “get” API function, which allows for the retrieval of metadata syntax currently being used by the Messaging LFB (see Section 5.2).

The operation of the Messaging LFB allows the SAE to instantiate an instance of the NPF Messaging specification that is specific to a particular application. The SAE is the one that determines what metadata requires a standard format (NPF Messaging format) for transport within an application. How the SAE goes about instantiating the operation of the Messaging LFB will vary from physical device to physical device. In some instances (i.e. for an ASIC) the instantiation of the Messaging LFB operation occurs when the SAE decides to purchase ASIC X that supports NPF Messaging format Y. In other cases (i.e. for a programmable NPU) the SAE may instantiate the Messaging LFB operation via the optional “set” function provided by the LFB or by setting a compile time option.

In simple terms, the system integrator should insert a Messaging LFB whenever a standard syntactic representation of metadata is required between two LFBs. This is typically observed when two functional LFBs span two physical entities (FEs). If there is no need for two LFBs to pass metadata, or if it’s acceptable for two LFBs to exchange metadata in their native format, then the Messaging LFB does not need to be inserted at that point in the Topology.

It should be noted that the Messaging LFB can act as a sink or source type of LFB. In other words, the Messaging LFB may be seen in an LFB topology without inputs, or without outputs, when it’s modeled at the ends of the topology. The Messaging LFB may also be used in conjunction with a sink/source type LFB other than the Messaging LFB.
Figure 2 illustrates how two distinct vendors might go about implementing the Messaging LFB. Consider Vendor X to be an ASIC vendor who manually sorts through the NPF Messaging IA and determines how metadata produced by the ASIC should be formatted to comply with the specification. Once this determination has been made and burned into the ASIC, that ASIC will henceforth be NPF Messaging compliant. In order for Vendor Y’s NPU to interoperate with the ASIC, the NPU code may need to be modified to expect the metadata syntax produced by the Vendor X ASIC. Alternatively, the NPU may support compile time options, and/or the Messaging LFBs “set” routine, which allow the system integrator to tell the NPU the metadata syntax to expect from the ASIC. Whatever the case, in this scenario the onus is on the NPU to adjust its syntactical representation of metadata to be able to interoperate with the hard-wired ASIC.

The Messaging LFB may optionally provide the capability of encoding/decoding different Messaging formats to frames sent and received by the LFB, based on a particular piece of metadata (e.g. apply Messaging format X for IPv4 packets and Messaging format Y for IPv6 packets). If the Messaging LFB supports encoding different Messaging formats to frames being sent out of the LFB, it must write the Messaging formats’ profile ID into metadata sent with the frame. Similarly, a Messaging LFB that supports the ability to decode different Messaging formats, based on a piece of metadata, must be able to read a piece of metadata from the received frame and verify that the content matches the profile ID of one of it’s configured Messaging formats. The ability to select which metadata element will be used in this process is a function of the LFBs API. The application, or SAE, will configure this functionality via the LFBs Set API function. If an instance of the Messaging LFB does not support these features, the relevant parameters of the API will be ignored.

The Messaging LFB supports the reception and transmission of any metadata elements specified by NPF LFB implementation agreements. Appendix A provides a sample header file which includes informative definitions of metadata elements. Normative definitions of metadata, however, are found among the various LFB implementation agreements defined by the NPF. Therefore, the implementer of the Messaging LFB will need to construct a header file that contains the appropriate list of normative metadata definitions, defined by NPF LFB implementation agreements, required by the application. The implementer may choose to use the header file in Appendix A as a basis for creating such a header file. The following diagram provides a pictorial representation of the Messaging LFB.
3.1 **LFB Input Ports**

The Messaging LFB supports two input ports which may be used to format received metadata from both sides (ingress and egress) of the LFB. One input port is used to receive a native metadata syntax which will be converted into an NPF Messaging compliant syntax. This input port is called \texttt{ntv\_in}. The other input port is used to receive an NPF Messaging compliant metadata syntax that will be converted into a metadata syntax native to the underlying device. This input port is called \texttt{msg\_in}.

3.1.1 Metadata Required

It is expected that the Messaging LFB will have access to all metadata generated, and not consumed, by the upstream LFBs that reside on the same physical entity (FE). However, it may not be necessary or expedient to transfer all metadata on the “wire”. It should be noted that the user may prohibit the number of metadata elements that are transmitted on the “wire” via the use of the Messaging LFBs \texttt{NPF\_F\_MSG\_SyntaxProfileSet} API function. This function allows you to, essentially, construct the stream of metadata elements that will be transmitted out of the Messaging LFB. This capability allows for certain optimizations with respect to the amount of actual data transferred on the “wire”.

If an instance of the Messaging LFB supports the ability to interpret different Messaging formats based on some piece of received metadata, it is expected that the application, or SAE, will be responsible for correlating which upstream metadata element will be used to select the Messaging format applied by the Messaging LFB. Once that determination is made, the application, or SAE, will communicate to the LFB which metadata element to look for via the LFBs API. The ability to interpret different Messaging formats only applies to frames coming into the \texttt{msg\_in} port of the LFB.

3.2 **LFB Output Ports**

The Messaging LFB supports two output ports which may be used to output packet and metadata from both sides (ingress and egress) of the LFB. One output port is used to transmit a native metadata syntax that has been decoded from an NPF Messaging compliant syntax. This output port is called \texttt{ntv\_out}. The other output port is used to transmit an NPF Messaging compliant metadata syntax that has been encoded from a metadata syntax native to the underlying device. This output port is called \texttt{msg\_out}. 
3.2.1 Metadata Produced

The Messaging LFB does not, in and of itself, create metadata. It simply passes through metadata it receives, altering only the syntactic representation of the metadata. The Messaging LFB does not alter the content of metadata that traverses through the LFB. The content of metadata passed in must be the same content passed out of the Messaging LFB. However, the Messaging LFB will alter the syntax of metadata so that it is NPF Messaging compliant. There are several syntactical elements that the Messaging LFB can alter.

- Location of a metadata element within the metadata stream
- Size of a metadata element

The Messaging LFB can modify these syntactic elements of metadata as long as those modifications do not change the content of the metadata (e.g. reduce the size of a metadata element to align with a byte granule without truncating any significant bits). In order to be compliant with NPF Messaging, the Messaging LFB must format all metadata elements, or at least verify compliance of all metadata elements.

If an instance of the Messaging LFB supports the ability to transmit frames using different Messaging formats, that instance must write a piece of metadata (configurable through the API) with the profile ID that is used to identify the Messaging format applied. Correlation of profile ID with Messaging format is a task for the application, and/or SAE. The ability to send frames of different Messaging formats only applies to frames being sent out on the “msg_out” port of the LFB.

It is important to note that the Messaging specification inherently contributes to the syntactic formatting of metadata. For instance, the Messaging specification defines a particular bit/byte order that should be adhered to. As such, not all syntactic attributes of metadata are controlled via the Messaging LFB. Only the attributes supported in the list above are controlled via the Messaging LFB.

3.3 Relationship with other LFBs

It is possible that the Messaging LFB will be seen in pairs in the forwarding plane. One Messaging LFB will be used for encoding metadata into the NPF Messaging format, while the other Messaging LFB will be used to decode the NPF Messaging compliant metadata. The Messaging LFB contains no intelligence with respect to the type of packet processing being done in the forwarding plane by other LFBs. The Messaging LFB is a transparent function, with respect to packet processing. Figure 4 presents a hypothetical example meant to illustrate various points in the forwarding plane where the Messaging LFB might be implemented.
*Assumption: ASICs in the diagram are NPF Messaging compliant

Figure 4 -- How the Messaging LFB might be used
4 Data Types

The Messaging LFB exposes an optional API which requires a few specific data types, defined in the following sub-sections.

4.1 Common LFB Data Types

4.1.1 LFB Type Code

The LFB type code for the Messaging LFB shall be 14.

#define NPF_LFB_TYPE_MESSAGING 14

4.1.2 Metadata enumeration

The implementation of the Messaging LFB will include a header file (.h file) which must include a typedef that is used to describe the metadata elements supported by the Messaging LFB instance. Appendix A provides an informative example of how the typedef might be specified. However, the normative definitions of metadata are contained in various NPF LFB implementation agreements. The Messaging LFB implementation must contain these normative definitions, and therefore the implementer must construct the metadata elements typedef out of the normative metadata definitions provided by the LFB implementation agreements. The typedef should be an enumeration of supported metadata elements by the Messaging LFB instance, and will be referenced by the metadata element structure defined in Section 4.1.2.

typedef NPF_uint8_t NPF_F_Metadata_Id_t;

4.1.3 Metadata element structure

The metadata element structure defines the attributes of a metadata element relevant to the Messaging LFB. NOTE: The first element of this structure (meta_id) is of the type specified in Section 4.1.1, named here as NPF_F_Metadata_Id_t.

typedef struct {
    NPF_F_Metadata_Id_t meta_id;         /* Id of metadata */
    NPF_uint8_t meta_offset;             /* Location of metadata */
    NPF_uint8_t meta_size;               /* Size of metadata (bytes) */
    NPF_boolean_t meta_trailer;          /* Specifies whether the */
                                            /* metadata is contained in */
                                            /* the trailer part of the */
                                            /* PDU. Otherwise, it is in */
                                            /* the header. */
} NPF_F_MetadataFormat_t;

4.1.4 Messaging Action

The Messaging action structure is used by the NPF_F_MSGMetaSelectorSet and NPF_F_MSGMetaSelectorGet functions. It specifies a read or write action.

typedef enum {
    NPF_MSG_READ = 0,
    NPF_MSG_WRITE = 1
} NPF_F_MSGAction_t;
4.1.5 Metadata Profile

The metadata profile structure is used to specify a list of metadata elements and the formatting of those elements. The structure also supports the optional feature of associating the Messaging format with a profile ID (profile_id). Instances of the Messaging LFB that support multiple Messaging formats will use the profile ID to distinguish between different Messaging formats configured in the LFB. The profile ID is also the value that a particular metadata must match in order for the Messaging format associated with the profile ID to be applied.

```c
typedef struct {
    NPF_uint8_t     num_meta_elements;
    NPF_F_MetadataFormat_t *meta_elements;
    NPF_uint8_t     profile_id;
} NPF_F_MSGMetaProfile_t;

typedef struct {
    NPF_uint8_t     num_profiles;
    NPF_F_MSGMetaProfile_t *profile;
} NPF_F_MSG_MetaProfileList_t;
```

4.2 Data Structures for Completion Callbacks

4.2.1 Asynchronous Response

```c
typedef struct {
    NPF_F_MSGReturnCode_t returnCode;
    union {
        NPF_F_MSGMetaProfile_t metaProfile;
        NPF_F_Metadata_Id_t meta_selector;
        NPF_F_MSG_MetaProfileList_t profile_list;
    } u;
} NPF_F_MSGAsyncResponse_t;
```

4.2.2 Callback Type

The callback response will contain one of the following codes that indicates the function that triggered the callback.

```c
typedef enum {
    NPF_F_MSG_SYNTAXPROFILE_SET = 1,
    NPF_F_MSG_SYNTAXPROFILE_GET = 2,
    NPF_F_MSG_SYNTAXALLPROFILES_GET = 3,
    NPF_F_MSG_METASELECTOR_SET = 4,
    NPF_F_MSG_METASELECTOR_GET = 5,
} NPF_F_MSGCallbackType_t;
```

<table>
<thead>
<tr>
<th>Callback Type</th>
<th>Callback Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPF_F_MSG_SYNTAXPROFILE_SET</td>
<td>NPF_F_MSGMetaProfile_t</td>
</tr>
<tr>
<td>NPF_F_MSG_SYNTAXPROFILE_GET</td>
<td>NPF_F_MSGMetaProfile_t</td>
</tr>
<tr>
<td>NPF_F_MSG_METASELECTOR_SET</td>
<td>n/a</td>
</tr>
<tr>
<td>NPF_F_MSG_METASELECTOR_GET</td>
<td>NPF_F_Metadata_Id_t</td>
</tr>
<tr>
<td>NPF_F_MSG_SYNTAXALLPROFILES_GET</td>
<td>NPF_F_MSG_MetaProfileList_t</td>
</tr>
</tbody>
</table>

Figure 4 -- Mapping of Callback Type to Callback Data
4.2.3 Callback Data

typedef struct {
    NPF_F_MSGCallbackType_t cbType;
    NPF_boolean_t allOk;
    NPF_uint32_t numResp;
    NPF_F_MSGAsyncResponse_t *resp;
} NPF_F_MSGCallbackData_t;

4.3 Data Structures for Event Notifications

Not applicable.

4.4 Error Codes

A typedef is defined to capture error codes returned by Messaging LFB functions.

typedef NPF_uint32_t NPF_F_MSGReturnCode_t;

The following sub-sections provide a list of error codes that could be returned by the API.

4.4.1 Common NPF Error Codes

#define NPF_NO_ERROR
#define NPF_E_UNKNOWN
#define NPF_E_BAD_CALLBACK_HANDLE
#define NPF_E_BAD_CALLBACK_FUNCTION
#define NPF_E_CALLBACK_ALREADY_REGISTERED

4.4.2 Messaging LFB Specific Error Codes

#define NPF_MSG_ERR(n) ((NPF_F_MSGReturnCode_t)NPF_MSG_ERR + (n))
#define NPF_E_MSG_META_NOT_SUPPORTED NPF_MSG_ERR(0)
#define NPF_E_MSG_INVALID_META_ELEMENT NPF_MSG_ERR(1)
#define NPF_E_MSG_BAD_FE_HANDLE NPF_MSG_ERR(2)
#define NPF_E_MSG_BAD_LFB_HANDLE NPF_MSG_ERR(3)
#define NPF_E_MSG_META_OVERWRITE NPF_MSG_ERR(4)
#define NPF_E_MSG_META_BAD_SIZE NPF_MSG_ERR(5)
#define NPF_E_MSG_META_ILLEGAL_OFFSET NPF_MSG_ERR(6)
#define NPF_E_MSG_BAD_PROFILE_ID NPF_MSG_ERR(7)
#define NPF_E_MSG_NOT_SUPPORTED NPF_MSG_ERR(8)
#define NPF_E_MSG_INVALID_ACTION NPF_MSG_ERR(9)
#define NPF_E_MSG_NO_PROFILES_DEFINED NPF_MSG_ERR(10)
5 Functional APIs (FAPIs)

5.1 Required APIs
The Messaging LFB does not define any required API functions.

5.2 Optional APIs
The Messaging LFB provides the following optional API functions:

- \texttt{NPF\_F\_MSGRegister()}: Allows the application to register a completion callback function with the Messaging FAPI implementation.
- \texttt{NPF\_F\_MSGDeregister()}: Allows the application to de-register the completion callback function with the Messaging FAPI implementation.
- \texttt{NPF\_F\_MSGSyntaxProfileGet()}: Returns a metadata syntax profile currently in use by the Messaging LFB.
- \texttt{NPF\_F\_MSGSyntaxAllProfilesGet()}: Returns all metadata syntax profiles currently configured in the Messaging LFB.
- \texttt{NPF\_F\_MSGSyntaxProfileSet()}: Allows for the configuration of a metadata syntax profile within the Messaging LFB.
- \texttt{NPF\_F\_MSGMetaSelectorSet()}: Used to tell the Messaging LFB which metadata element to read or write to determine how to encode or decode a Messaging format.
- \texttt{NPF\_F\_MSGMetaSelectorGet()}: Returns a numeric value that represents the metadata element the LFB is currently reading or writing.

5.2.1 Completion Callback Function

\begin{verbatim}
typedef void (*NPF\_F\_MSGCallBackFunc\_t)(
    NPF\_IN NPF\_userContext\_t userContext,
    NPF\_IN NPF\_correlator\_t correlator,
    NPF\_IN NPF\_F\_MSGCallbackData\_t msgCallbackData);
\end{verbatim}

5.2.1.1 Description
This callback function is used by the application to register an asynchronous response handling routine with the NPF Messaging Functional API implementation. This callback function is intended to be implemented by the application, and registered with the NPF Messaging Functional API implementation through the \texttt{NPF\_F\_MSGRegister()} function.

5.2.1.2 Input Parameters
- \texttt{userContext} – The context item that was supplied by the application when the completion callback function was registered.
- \texttt{correlator} – The correlator item that was supplied by the application when the Messaging Functional API call was made. The correlator is used by the application mainly to distinguish between multiple invocations of the same function.
- \texttt{msgCallbackData} – Response information related to the Messaging functional API function call. This structure contains the syntactic details of the metadata used by the Messaging LFB. See Section 4.2 for more details.

5.2.1.3 Output Parameters
None.
5.2.1.4 Immediate Return Codes
None.

5.2.1.5 Callback Response
Not applicable.

5.2.1.6 Notes
None.

5.2.2 NPF_F_MSGRegister()

NPF_error_t NPF_F_MSGRegister(
    NPF_IN NPF_userContext_t userContext,
    NPF_IN NPF_F_MSGCallBackFunc_t msgCallbackFunc,
    NPF_OUT NPF_callbackHandle_t *msgCallbackHandle);

5.2.2.1 Description
This function is used by an application to register its completion callback function for receiving
asynchronous responses related to NPF Messaging FAPI function calls. The application may register
multiple callback functions using this function. The callback function is identified by the pair of
userContext and msgCallbackFunc, and for each individual pair, a unique msgCallbackHandle will be
assigned for future reference. Since the callback function is identified by both userContext and
msgCallbackFunc, duplicate registration of same callback function with different userContext is
allowed. Also, the same userContext can be shared among different callback functions. Duplicate
registration of the same userContext and msgCallbackFunc pair has no effect, and will output a
handle that is already assigned to the pair, and will return
NPF_F_MSG_E_ALREADY_REGISTERED.
Note: NPF_F_MSGRegister( ) is a synchronous function and has no completion callback
associated with it.

5.2.2.2 Input Parameters
- userContext – A context item used 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 1st parameter when it is called. The application
can assign any value to the userContext and the value is completely opaque to the NPF
  Messaging functional API implementation.
- msgCallbackFunc – The pointer to the completion callback function to be registered.

5.2.2.3 Output Parameters
- msgCallbackHandle – A unique identifier assigned for the registered userContext and
  msgCallbackFunc pair. This handle will be used by the application to specify which callback
  function to be called when invoking asynchronous NPF Messaging FAPI functions. It will also
  be used when de-registering the userContext and msgCallbackFunc pair.

5.2.2.4 Immediate Return Codes
- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_BAD_CALLBACK_FUNCTION**: msgCallbackFunc is NULL.
- **NPF_E_CALLBACK_ALREADY_REGISTERED:** No new registration was made since the userContext and msgCallbackFunc pair were already registered. Note: Whether this should be treated as an error or not is dependent on the application.

### 5.2.2.5 Callback Response

Not applicable.

### 5.2.2.6 Notes

None.

### 5.2.3 NPF_F_MSGDeregister()

```c
NPF_error_t NPF_F_MSGDeregister(
    NPF_IN NPF_callbackHandle_t msgCallbackHandle);
```

#### 5.2.3.1 Description

This function is used by an application to de-register a pair of user context and callback function. Note: If there are any outstanding calls related to the de-registered callback function, the callback function may be called for those outstanding calls even after de-registration. Note: **NPF_F_MSGDeregister()** is a synchronous function and has no completion callback associated with it.

#### 5.2.3.2 Input Parameters

- `msgCallbackHandle` – The unique identifier representing the pair of user context and callback function to be de-registered.

#### 5.2.3.3 Output Parameters

None.

#### 5.2.3.4 Immediate Return Codes

- **NPF_NO_ERROR:** Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN:** Returned if the function encounters an error of unknown type.
- **NPF_E_BAD_CALLBACK_HANDLE:** The API implementation does not recognize the callback handle. There is no effect to the registered callback functions.

#### 5.2.3.5 Callback Response

Not applicable.

#### 5.2.3.6 Notes

None.

### 5.2.4 NPF_F_MSGSyntaxProfileSet()

```c
NPF_error_t NPF_F_MSGSyntaxProfileSet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
```
5.2.4.1 Description

This routine allows for the configuration of a metadata syntax profile of the Messaging LFB. The application passes in a list of metadata characteristics (passed in via *metaProfile) that should be used to alter the syntax of the current metadata stream produced / consumed by the Messaging LFB. The routine will return, via the completion callback function, a list of metadata characteristics which reflects what the implementation actually set.

By virtue of the metadata elements passed in via the metaList parameter, this function can restrict the number of metadata elements that are sent out on the “wire” by this LFB. If the user does not want a particular metadata element to be sent out, then that metadata element should not be specified in the metaProfile parameter.

If the application is using an instance of the Messaging LFB that supports multiple metadata syntax profiles, then a profile ID must be associated with the list of metadata elements. Both the profile ID and the metadata elements are contained within the metaProfile parameter. To change a profile that has already been configured, the user should call this function using the profile ID of the profile that is to be changed. Otherwise, if the profile ID does not match any of the profile IDs of the profiles that have already been configured, a new profile will be created.

5.2.4.2 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 call.
- **feHandle**: The handle of the forwarding element in which the Messaging LFB is located.
- **lfbId**: The ID of the Messaging LFB.
- **metaProfile**: A pointer to the NPF_F_MSGMetaProfile_t structure which contains a profile Id, a list of metadata elements and their configurable attributes.

5.2.4.3 Output Parameters

None.

5.2.4.4 Immediate Return Codes

- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_MSG_BAD_FE_HANDLE**: Returned if an invalid FE handle is passed into the routine.
- **NPF_E_MSG_BAD_LFB_HANDLE**: Returned if an invalid LFB handle is passed into the routine.

5.2.4.5 Callback Response

The callback response for this function will provide to the application the list of metadata elements (via the NPF_F_MSGCallbackData_t structure) that were successfully configured as specified, along with one of the return codes specified below. The application may use the returned structure to determine, in the event that the function failed, which metadata element(s) caused the problem.
- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_MSG_META_NOT_SUPPORTED**: Returned if the underlying implementation does not support one, or more, of the metadata elements specified in *metaList.
- **NPF_E_MSG_INVALID_META_ELEMENT**: Returned if one of the attributes of a metadata element is not be specified for that metadata.
- **NPF_E_MSG_META_OVERWRITE**: Returned if more than one metadata element has been specified to reside in the same offset location.
- **NPF_E_MSG_META_BAD_SIZE**: Returned if a particular metadata element cannot be configured to be of the size specified.
- **NPF_E_MSG_META_ILLEGAL_OFFSET**: Returned if a particular metadata element cannot be placed in the offset specified.
- **NPF_E_MSG_BAD_PROFILE_ID**: Returned if the profile ID passed in is invalid.

### 5.2.4.6 Notes

None.

### 5.2.5 NPF_F_MSGSyntaxProfileGet()

```c
NPF_error_t NPF_F_MSGSyntaxProfileGet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId,
    NPF_IN NPF_uint8_t profileId);
```

#### 5.2.5.1 Description

This routine returns, via a completion callback function, a metadata syntax profile currently accessible by the Messaging LFB. For instances of the Messaging LFB that do not support multiple Messaging syntax profiles, the `profileId` parameter is effectively ignored by the implementation of this function.

#### 5.2.5.2 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 call.
- **feHandle**: The handle of the forwarding element in which the Messaging LFB is located.
- **lfbId**: The ID of the Messaging LFB.
- **profileId**: The ID of the metadata syntax profile to be retrieved.

#### 5.2.5.3 Output Parameters

None.

#### 5.2.5.4 Immediate Return Codes

- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_MSG_BAD_FE_HANDLE**: Returned if an invalid FE handle is passed into the routine.
- **NPF_E_MSG_BAD_LFB_HANDLE**: Returned if an invalid LFB handle is passed into the routine.

### 5.2.5.5 Callback Response

The callback response for this function will provide to the application the list of metadata elements (via the `NPF_F_MSGCallbackData_t` structure) assigned to the specified profile ID. The return code field of the `NPF_F_MSGCallbackData_t` structure may contain one of two return codes:

- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_MSG_BAD_PROFILE_ID**: Returned if the profile ID passed in does not match any of the currently configured profile IDs.
- **NPF_E_MSG_NO_PROFILES_DEFINED**: Returned if there are currently no metadata syntax profiles configured in the Messaging LFB.

### 5.2.5.6 Notes

None.

### 5.2.6 NPF_F_MSGSyntaxAllProfilesGet()

```c
NPF_error_t NPF_F_MSGSyntaxAllProfilesGet(  
  NPF_IN NPF_callbackHandle_t  callbackHandle,  
  NPF_IN NPF_correlator_t     correlator,  
  NPF_IN NPF_errorReporting_t errorReporting,  
  NPF_IN NPF_FEHandle_t       feHandle,  
  NPF_IN NPF_LFB_ID_t         lfbId);
```

#### 5.2.6.1 Description

This routine returns, via a completion callback function, all metadata syntax profiles currently configured in the Messaging LFB. For instances of the Messaging LFB that do not support multiple Messaging syntax profiles, this function will return a single profile (if configured).

#### 5.2.6.2 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 call.
- **feHandle**: The handle of the forwarding element in which the Messaging LFB is located.
- **lfbId**: The ID of the Messaging LFB.

#### 5.2.6.3 Output Parameters

None.

#### 5.2.6.4 Immediate Return Codes

- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_MSG_BAD_FE_HANDLE**: Returned if an invalid FE handle is passed into the routine.
- **NPF_E_MSG_BAD_LFB_HANDLE**: Returned if an invalid LFB handle is passed into the routine.

### 5.2.6.5 Callback Response

The callback response for this function will provide to the application the list of metadata elements (via the `NPF_F_MSGCallbackData_t` structure) assigned to the specified profile ID. The return code field of the `NPF_F_MSGCallbackData_t` structure may contain one of two return codes:

- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_MSG_NO_PROFILES_DEFINED**: Returned if there are currently no metadata syntax profiles configured in the Messaging LFB.

### 5.2.6.6 Notes

None.

### 5.2.7 `NPF_F_MSGMetaSelectorSet ()`

```c
NPF_error_t NPF_F_MSGMetaSelectorSet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId,
    NPF_IN NPF_F_Metadata_Id_t meta_selector,
    NPF_IN NPF_F_MSGAction_t meta_action);
```

#### 5.2.7.1 Description

This routine is used to tell the Messaging LFB which metadata element to read or write, in order to determine how to encode or decode a Messaging format. If the `meta_action` field is set for reading, this function will tell the LFB which metadata to read, on the “msg_in” port, in order to determine the Messaging format to apply to the incoming frame. If the `meta_action` field is set for writing, this function will tell the LFB which metadata to write, on the “msg_out” port, the profile ID of the Messaging format being applied to outgoing frames.

Note that the purpose of the metadata selector is to specify which metadata element the LFB should be looking for as a message (an NPF Messaging compliant frame, which contains packet and embedded metadata) comes into the LFB. The LFB will in turn read the contents of that piece of metadata and try to match it against one of the profile IDs configured in the LFB. If it finds a match, it applies the formatting of that profile to the message.

#### 5.2.7.2 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 call.
- **feHandle**: The handle of the forwarding element in which the Messaging LFB is located.
- **lfbId**: The ID of the Messaging LFB.
- **meta_selector**: A numeric value that represents the metadata element that the Messaging LFB needs to read or write.
- **meta_action**: Specifies whether the metadata selection applies to the reading of frames (frames coming in on port MSG_IN) or the writing of frames (frames going out on port MSG_OUT).

### 5.2.7.3 Output Parameters

None.

### 5.2.7.4 Immediate Return Codes

- **NPF_NO_ERROR**: Returned if no errors encountered by the function.
- **NPF_E_UNKNOWN**: Returned if the function encounters an error of unknown type.
- **NPF_E_MSG_BAD_FE_HANDLE**: Returned if an invalid FE handle is passed into the routine.
- **NPF_E_MSG_BAD_LFB_HANDLE**: Returned if an invalid LFB handle is passed into the routine.
- **NPF_E_MSG_NOT_SUPPORTED**: Returned if the instance of the LFB does not support multiple Messaging formats.
- **NPF_E_MSG_INVALID_ACTION**: Returned if the action specified is invalid.

### 5.2.7.5 Callback Response

None.

### 5.2.7.6 Notes

This function is only used for instances of the Messaging LFB that support multiple Messaging formats.

### 5.2.8 NPF_F_MSGMetaSelectorGet ()

```c
NPF_error_t NPF_F_MSGMetaSelectorGet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId,
    NPF_IN NPF_F_MSGAction_t meta_action);
```

#### 5.2.8.1 Description

This routine returns a numeric value that represents the metadata element the LFB is currently reading or writing. If the meta_action field is set for reading, this function will return the metadata element that the LFB reads to determine the Messaging format to apply on incoming frames. If the meta_action field is set for writing, this function will return the metadata element that the LFB writes the profile ID to, for frames going out of port MSG_OUT.

Note that the purpose of the metadata selector is to specify which metadata element the LFB should be looking for as a message (an NPF Messaging compliant frame, which contains packet and embedded metadata) comes into the LFB. The LFB will in turn read the contents of that piece of metadata and try to match it against one of the profile IDs configured in the LFB. If it finds a match, it applies the formatting of that profile to the message.
5.2.8.2 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 call.
- `feHandle`: The handle of the forwarding element in which the Messaging LFB is located.
- `lfbId`: The ID of the Messaging LFB.
- `meta_action`: Specifies whether to return the metadata selector for the reading of frames (frames coming in on port `MSG_IN`) or the writing of frames (frames going out on port `MSG_OUT`).

5.2.8.3 Output Parameters

None.

5.2.8.4 Immediate Return Codes

- `NPF_NO_ERROR`: Returned if no errors encountered by the function.
- `NPF_E_UNKNOWN`: Returned if the function encounters an error of unknown type.
- `NPF_E_MSG_BAD_FE_HANDLE`: Returned if an invalid FE handle is passed into the routine.
- `NPF_E_MSG_BAD_LFB_HANDLE`: Returned if an invalid LFB handle is passed into the routine.
- `NPF_E_MSG_NOT_SUPPORTED`: Returned if the instance of the LFB does not support multiple Messaging formats.
- `NPF_E_MSG_INVALID_ACTION`: Returned if the action specified is invalid.

5.2.8.5 Callback Response

The callback response for this function will provide to the application the current metadata selector. The return code field of the `NPF_F_MSGCallbackData_t` structure may contain one of two return codes:

- `NPF_NO_ERROR`: Returned if no errors encountered by the function.
- `NPF_E_UNKNOWN`: Returned if the function encounters an error of unknown type.

5.2.8.6 Notes

This function is only used for instances of the Messaging LFB that support multiple Messaging formats.
6 References


APPENDIX A  NPF_F_MSG.H

/*
 * This header file defines typedefs, constants, and structures
 * for the NP Forum Messaging Functional API
 */
#ifndef __NPF_F_MSG_H__
#define __NPF_F_MSG_H__

#ifdef __cplusplus
extern "C" {
#endif

#define NPF_LFB_TYPE_MESSAGING 14

typedef enum {
  NPF_META_HEADER_TYPE = 0,  /* Type of header (i.e. IPv4, IPv6, etc.) */
  NPF_META_TIMESTAMP = 1,     /* Specifies a system tick from the transmitting NPE at the time the packet was queued for transmission. */
  NPF_META_SAP_ID = 2,  /* Specifies the application endpoint of the message. */
} NPF_META_ENUM;

typedef enum {
  NPF_META_HEADER_TYPE = 0,
  NPF_META_TIMESTAMP = 1,
  NPF_META_SAP_ID = 2,
} NPF_META_ENUM;

typedef enum {
  NPF_META_HEADER_TYPE = 0,  /* Type of header (i.e. IPv4, IPv6, etc.) */
  NPF_META_TIMESTAMP = 1,     /* Specifies a system tick from the transmitting NPE at the time the packet was queued for transmission. */
  NPF_META_SAP_ID = 2,  /* Specifies the application endpoint of the message. */
} NPF_META_ENUM;

typedef enum {
  NPF_META_HEADER_TYPE = 0,  /* Type of header (i.e. IPv4, IPv6, etc.) */
  NPF_META_TIMESTAMP = 1,     /* Specifies a system tick from the transmitting NPE at the time the packet was queued for transmission. */
  NPF_META_SAP_ID = 2,  /* Specifies the application endpoint of the message. */
} NPF_META_ENUM;
NPF_META_CONTEXT_ID = 3,            /* Specifies the application-  
* endpoint-specific state of  
* the message. */
NPF_META_FORMAT_ID = 4,              /* Specifies which format to use  
* to parse the remaining  
* metadata string */
NPF_META_OFFSET = 5,                 /* Specifies an offset, in bytes,  
* into the message payload. */
NPF_META_CRC_32 = 6,                 /* Determines the integrity of all  
* bytes in the message up to the  
* integrity field, consistent  
* with ISO 3309. */
NPF_META_CKSUM_16 = 7,               /* Determines the integrity of all  
* bytes in the message up to the  
* integrity field, consistent  
* with RFC1071 and RFC1141. */
NPF_META_RX_STAT = 8,                /* Receive status flag (Unicast,  
* Broadcast, Multicast, etc.) */
NPF_META_PACKET_SIZE = 9,            /* Total packet size */
NPF_META_OUTPUT_PORT = 10,           /* Egress output port */
NPF_META_INPUT_PORT = 11,            /* Ingress input port */
NPF_META_FABRIC_PORT = 12,           /* Output port for fabric  
* indicating the blade ID */
NPF_META_FLOW_ID = 13,               /* QOS flow id or MPLS label/flow  
* id */
NPF_META_FLOW_QOS = 14,              /* Specifies the quality of  
* service associated with the  
* flow, as defined by the  
* application. */
NPF_META_FLOW_STATE = 15,            /* Determined by interoperating  
* vendors to meet the  
* requirements of the system  
* architecture */
NPF_META_NEXTHOP_ID = 16,            /* Next hop IP ID */
NPF_META_CLASS_ID = 17,              /* Class ID */
NPF_META_COLOR = 18,                 /* Color */
NPF_META_NPE_TABLE = 19,             /* Specifies the "table" within  
* the NPE that the npe-operation  
* is targeting. */
NPF_META_NPE_REGISTER = 20,          /* Specifies the "register" within  
* the NPE that the npe-operation  
* is targeting. */
NPF_META_NPE_RECORD = 21,            /* Specifies the "record" within  
* an npe-table that the  
* npe-operation is targeting. */
NPF_META_NPE_OPERATION = 22, /* Specifies a common set of * operations that an NPE can * perform. */
NPF_META_SEND_SEQUENCE = 23, /* Specifies an incrementing * number of messages (starting * from 0), relative to the flow, * that has been generated * by the sending NPE. */
NPF_META_ACK_SEQUENCE = 24, /* Specifies the last Sequence * Number processed by the * receiving NPE relative to the * flow. */
NPF_META_USER_ATTRIBUTE = 25, /* Determined by interoperating * vendors to meet the * requirements of the system * architecture. */
NPF_META_CHANNEL_ID = 26, /* Channel ID */
NPF_META_INGRESS_CHANNEL = 27, /* Ingress Channel number */
NPF_META_EGRESS_CHANNEL = 28, /* Egress Channel number */
NPF_META_END_META_LIST = 29 /* Always the last element of this * enumeration */
}
NPF_F_Metadata_Id_t;

typedef enum {
    NPF_F_MSG_SYNTAXPROFILE_SET = 1,
    NPF_F_MSG_SYNTAXPROFILE_GET = 2,
    NPF_F_MSG_SYNTAXACCESS_PROFILE_GET = 3,
    NPF_F_MSG_METASELECTOR_SET = 4,
    NPF_F_MSG_METASELECTOR_GET = 5,
} NPF_F_MSGCallbackType_t;

typedef struct {
    NPF_F_Metadata_Id_t meta_id; /* Id of metadata */
    NPF_uint8_t meta_offset; /* Location of metadata */
    NPF_uint8_t meta_size; /* Size of metadata (bytes) */
    NPF_boolean_t meta_trailer; /* Specifies whether the * metadata is contained in * the trailer part of the * PDU. Otherwise, it is * in the header. */
} NPF_F_MetadataFormat_t;

typedef enum {
    NPF_MSG_READ = 0,
    NPF_MSG_WRITE = 1
} NPF_F_MSGAction_t;

typedef struct {
    /* Metadata Profile Structure */
    NPF_F_MetadataProfile_t;
    } NPF_F_MetadataFormat_t;

typedef enum {
    NPF_MSG_READ = 0,
    NPF_MSG_WRITE = 1
} NPF_F_MSGAction_t;
typedef struct {  
  NPF_uint8_t num_meta_elements;
  NPF_F_MetadataFormat_t *meta_elements;
  NPF_uint8_t profile_id;
} NPF_F_MSGMetaProfile_t;

/* Metadata profile list */
typedef struct {
  NPF_uint8_t num_profiles;
  NPF_F_MSGMetaProfile_t *profile;
} NPF_F_MSG_MetaProfileList_t;

/* Asynchronous Response */
typedef struct {
  NPF_F_MSGReturnCode_t returnCode;
  union {
    NPF_F_MSGMetaProfile_t metaProfile;
    NPF_F_Metadata_Id_t meta_selector;
    NPF_F_MSG_MetaProfileList_t profile_list;
  } u;
} NPF_F_MSGAsyncResponse_t;

/* Callback Data */
typedef struct {
  NPF_F(MSGCallbackType_t cbType;
  NPF_boolean_t allOk;
  NPF_uint32_t numResp;
  NPF_F_MSGAsyncResponse_t *resp;
} NPF_F_MSGCallbackData_t;

/* Function Prototypes */

/* Completion Callback Function */
typedef void (*NPF_F_MSGCallBackFunc_t)(
  NPF_IN NPF_userContext_t userContext,
  NPF_IN NPF_correlator_t correlator,
  NPF_IN NPF_F_MSGCallbackData_t msgCallbackData);

/* Callback Registration Function */
NPF_error_t NPF_F_MSGRegister(
  NPF_IN NPF_userContext_t userContext,
  NPF_IN NPF_F_MSGCallBackFunc_t msgCallbackFunc,
  NPF_OUT NPF_callbackHandle_t *msgCallbackHandle);

/* Callback De-Registration Function */
NPF_error_t NPF_F_MSGDeregister(
  NPF_IN NPF_callbackHandle_t msgCallbackHandle);

/* Messaging Syntax Set Function */
NPF_error_t NPF_F_MSGSyntaxProfileSet(
  NPF_IN NPF_callbackHandle_t callbackHandle,
NPF_IN NPF_correlator_t correlator,
NPF_IN NPF_errorReporting_t errorReporting,
NPF_IN NPF_FEHandle_t feHandle,
NPF_IN NPF_LFB_ID_t lfbId,
NPF_IN NPF_FErrorMsgMetaProfile_t *metaProfile);

/*
 * Messaging Syntax Get Function
 */
NPF_error_t NPF_F_MSGSyntaxProfileGet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId,
    NPF_IN NPF_uint8_t profileId);

/*
 * Messaging Syntax Get All Function
 */
NPF_error_t NPF_F_MSGSyntaxAllProfilesGet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId);

/*
 * Messaging Metadata Selector Set Function
 */
NPF_error_t NPF_F_MSGMetaSelectorSet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId,
    NPF_IN NPF_F_Metadata_Id_t meta_selector,
    NPF_IN NPF_F_MSGAction_t meta_action);

/*
 * Messaging Metadata Selector Get Function
 */
NPF_error_t NPF_F_MSGMetaSelectorGet(
    NPF_IN NPF_callbackHandle_t callbackHandle,
    NPF_IN NPF_correlator_t correlator,
    NPF_IN NPF_errorReporting_t errorReporting,
    NPF_IN NPF_FEHandle_t feHandle,
    NPF_IN NPF_LFB_ID_t lfbId,
    NPF_IN NPF_F_MSGAction_t meta_action);

#endif /* __cplusplus */
#endif

#endif /* __NPF_F_MSG_H__ */
APPENDIX B  LIST OF COMPANIES BELONGING TO NPF DURING APPROVAL PROCESS

<table>
<thead>
<tr>
<th>Company</th>
<th>Company</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agere Systems</td>
<td>HCL Technologies</td>
<td>Nokia</td>
</tr>
<tr>
<td>Altera</td>
<td>Hifn</td>
<td>Nortel Networks</td>
</tr>
<tr>
<td>AMCC</td>
<td>IBM</td>
<td>PMC Sierra</td>
</tr>
<tr>
<td>Analog Devices</td>
<td>IDT</td>
<td>Silicon &amp; Software Systems</td>
</tr>
<tr>
<td>Avici Systems</td>
<td>Intel</td>
<td>Sun Microsystems</td>
</tr>
<tr>
<td>Cypress Semiconductor</td>
<td>IP Fabrics</td>
<td>Teja Technologies</td>
</tr>
<tr>
<td>Ericsson</td>
<td>IP Infusion</td>
<td>TranSwitch</td>
</tr>
<tr>
<td>Erlang Technologies</td>
<td>ISIC Corporation</td>
<td>U4EA Group</td>
</tr>
<tr>
<td>ETRI</td>
<td>Kawasaki LSI</td>
<td>Xelerated</td>
</tr>
<tr>
<td>EZChip</td>
<td>Motorola</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Flextronics</td>
<td>NetLogic</td>
<td>Zettacom</td>
</tr>
<tr>
<td>FutureSoft</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ACKNOWLEDGEMENTS

Gamil Cain, Intel  
Zsolt Haraszti, Ericsson  
Sten Johnson, Ericsson  
Vinoj Kumar, Agere Systems  
David Maxwell, IDT  
Robert E. Penny, Ericsson  
John Renwick, Agere Systems