Generic Classifier LFB and Functional API Implementation Agreement

22 December 2004

Revision 1.0

Editor:

John Renwick, Agere Systems, jrenwick@agere.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 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

Table of Figures

1 Revision History

Revision	Date	Reason for Changes
1.0	10/29/2004	Created Rev 1.0 of the implementation agreement by taking the Generic Classifier LFB and API document (npf2002.478.16) and making editorial and minor technical corrections.

2 Introduction

This document describes a Classifier LFB. A Classifier examines a packet and, based on its content, produces an output ClassID metadata that can affect the processing of that packet in downstream LFBs.

2.1 Acronyms / Definitions

Terminology

- **PDU**: Protocol Data Unit. This is the portion of a packet beginning with the header of a certain protocol. A packet can contain multiple PDUs; for instance there might be a TCP PDU inside an IP PDU, which is inside a PPP PDU, which is inside an Ethernet MAC PDU (frame).
- **Metadata**: information associated with a packet, but not part of its contents. For example, an identifier of the interface a packet arrived on could be used as metadata.
- **Filter**: the specification of one or more fields of one or more PDUs in a packet to be examined.
- **Pattern**: a set of values to be compared with the packet fields indicated by the filter.
- **Action**: specification of what is to be done when a packet matches a pattern, consisting of these alternatives:
    - **ClassID**: a metadata value output by the filter
    - **NextFilter**: the identifier of the next filter (within the same LFB) to process the packet.
- **Rule**: a specific combination of a filter, pattern, and action. When a packet matches a certain rule, the action associated with the rule is taken.

2.2 Assumptions

2.3 Scope

In many network processor designs, the set of defined filters is established at the time the device is initialized; often it is implicit in microcode or other forms of instructions loaded into the device. In such devices, patterns and actions typically can be programmed freely at run-time without interrupting the device's operation, but the same is not true of filters. The NP Forum Software Working Group regards booting and initial configuration operations as outside the scope of its APIs. Accordingly, this agreement supports discovery of available filters, but not the run-time creation or reconfiguration of them. It does support run-time programming of patterns and actions, and thus the creation of new rules based on the device's existing repertoire of filters.

And it supports the "cloning" of filters, or creation of new filters with characteristics identical to existing ones, under the assumption that the implementation can manage discrete sets of matching patterns and actions for the same message content.

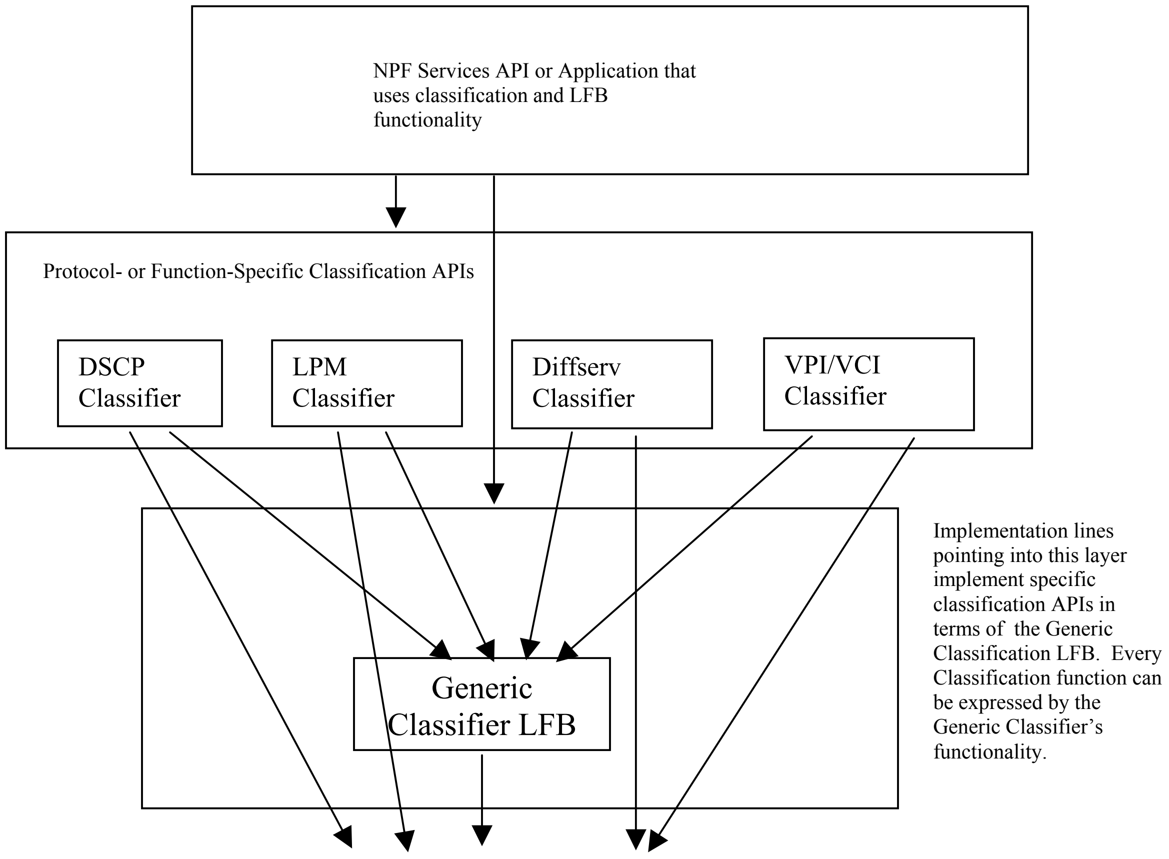
2.4 External Requirements and Dependencies

This agreement depends on the Protocol Type structure (**`NPF_protocol_t`**), first defined in the NPF Packet Handler Implementation Agreement, version 1.0, September, 2003.

3 LFB Detailed Description

Classification filters are resources of Classifier LFBs in the terms of the NP Forum Functional API (FAPI). The general function of classification is to examine a packet with its metadata, determine something about what kind of packet it is, and produce a ClassID, a metadata value that summarizes its finding. (The value is assigned to a metadata tag of **`NPF_META_CLASS_ID`** by default; however a Generic Classifier instance can assign the value to a different metadata class.) The basic tool of this API is the filter, which is a process that can locate and parse packet headers, and locate fields within headers. A filter examines one or more fields in a packet or metadata, compares them to known values, and acts based on which values match the packet fields.

Many protocols have requirements for particular kinds of filters, such as diffserv classifiers, destination address lookup tables, VPI/VCI maps, etc. This API assumes a "generic" filter capability, upon which the implementations of any of these can be based, but is not required to be based. Generic classifiers can also be used directly by applications.


NPF Services API or Application that uses classification and LFB functionality
Protocol- or Function-Specific Classification APIs
DSCP Classifier
LPM Classifier
Diffserv Classifier
VPI/VCI Classifier
Generic Classifier LFB
Implementation lines pointing into this layer implement specific classification APIs in terms of the Generic Classification LFB. Every Classification function can be expressed by the Generic Classifier's functionality.

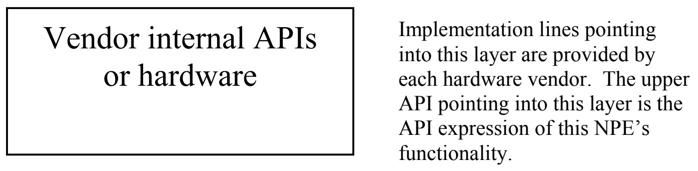

Vendor internal APIs or hardware
Implementation lines pointing into this layer are provided by each hardware vendor. The upper API pointing into this layer is the API expression of this NPE's functionality.

Figure 1: Relationship to other classifiers

We assume a forwarding device that supports filters has been "preloaded" with a set of filters, and the logic, configuration or microcode needed to process packets through those filters. This API allows the application to discover the nature and IDs for those filters, and to program their pattern and action values.

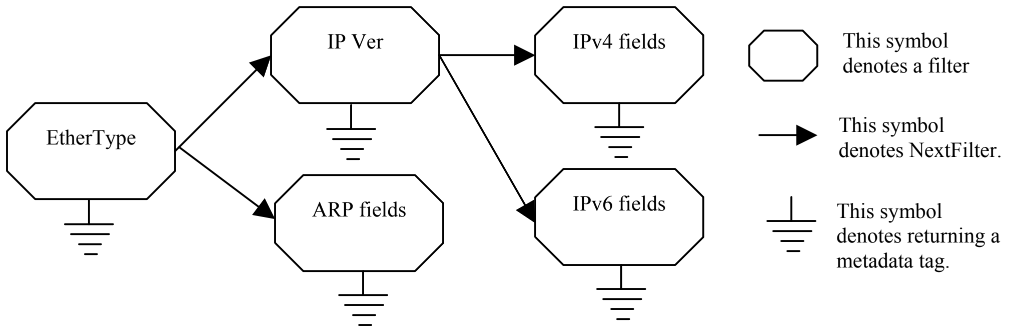

IP Ver
IPv4 fields
EtherType
ARP fields
IPv6 fields
This symbol denotes a filter
This symbol denotes NextFilter.
This symbol denotes returning a metadata tag.

Figure 2: Filter Chaining

<u>Filter Chaining</u>

In typical protocol header processing, a field in one protocol's header will indicate the protocol of the PDU that follows. Chaining of filters, in which the result of one filter can select the next filter to be applied (see Figure 2) facilitates this kind of processing. In this very simplified example, the FE is processing an Ethernet MAC PDU. The first filter examines the EtherType field, looking for IP (0x800), ARP (0x806), or some other code that it doesn't know how to process further. In the latter case, the rule may assign a special ClassID value to the packet, meaning "unknown or unsupported protocol". The rules for IP and ARP shift the focus to other filters specific to those PDU types.

Note: this is an example of one way of parsing a packet. Alternatively, this could be done by a multi-field filter that looks at key fields of all the possible PDUs in a single filter definition.

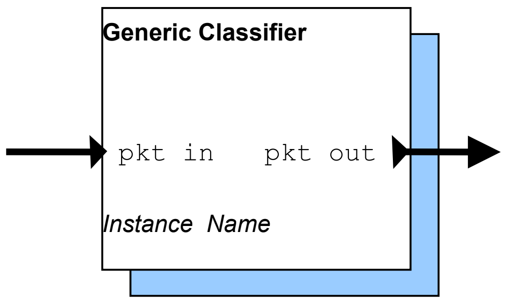


Figure 1: Generic LFB Diagram

3.1 Generic Classifier LFB Type Code

The LFB Type code for a Generic Classifier LFB shall be 13.

```
#define NPF_LFB_TYPE_GENERIC_CLASSIFIER 13
```

3.2 LFB Input Ports

The Generic Classifier LFB as one input port, PKT_IN, on which it receives packets to be classified.

3.2.1 Metadata Required

The Generic Classifier LFB can examine metadata as part of packet classification. For example, when classifying packets received from an external network interface, the identity of that interface can influence the classification result. The implementation determines what metadata is used for classification, and if any is used, the client is responsible for knowing how it is represented and what its values mean. The client causes metadata to be examined by specifying in filters what metadata to look at and what values to use for comparison.

3.3 LFB Output Ports

The Generic Classifier LFB has one output port PKT_OUT, on which it emits the packets it receives, unaltered, along with **`NPF_META_CLASS_ID`** (or other type) metadata.

3.3.1 Metadata Produced

The Generic Classifier LFB produces a ClassID value as metadata. By default, this value is assigned to a metadata class of **`NPF_META_CLASS_ID`**, but any Generic Classifier instance can assign its output value to a different metadata class. The application can find out what metadata class is produced by calling the **`NPF_F_gclassLFB_AttrGet()`** function.

3.4 Relationship with other LFBs

The Generic Classifier LFB can be preceded in the topology by any LFB type that emits a packet. Downstream (not necessarily next) there should be an LFB that consumes the ClassID (or other metadata tag) produced.

4 Data Types

4.1 Common LFB Data Types.

4.1.1 Filter ID

A filter is a resource of a FAPI Classifier Logical Function Block. The LFB assigns each filter an ID value that identifies it within the context of that LFB.

```
typedef NPF_uint32_t     NPF_FilterID_t;
```

4.1.2 Filter Class

Filters can classify based on a contiguous string of bits within the packet (single field filter), or on multiple such fields (multi-field filter). This difference establishes the “class” of a filter. In the filter definition structure, the number of fields indicates class: one, or more than one.

4.1.3 Filter Type

Each filter obeys a certain definition of the precedence of rules that use it, in case it is possible that more than one rule could match a given packet. The filter type identifies the method of establishing precedence between rules:

- No precedence – the rules are constructed such that no packet can match more than one rule. Flat table lookups and MAC address lookups are examples.
- Ordered precedence – rules are maintained in a specific order, and precedence belongs to the first rule that matches.
- LPM – Longest Prefix Match. When two patterns could match, the one with the least number of don’t care bits (or the greatest number of specific pattern bits) matches.

Note: a No Precedence filter is essentially the degenerate case of LPM in which all prefixes have the same length.

```
typedef enum {
    NPF_GCLASS_TYPE_NO_PRECEDENCE = 1,
    NPF_GCLASS_TYPE_ORDERED_PRECEDENCE = 2,
    NPF_GCLASS_TYPE_LPM = 3
} NPF_gclassFilterType_t;
```

4.1.4 Filter Attribute

Each filter can be either fixed (rules may not be changed), configurable (rules can be added and deleted) or clonable (configurable, and also a new filter can be created using this one as a pattern).

```
/*
 * Filter Attribute
 */
typedef enum      {
      NPF_GCLASS_FILTER_FIXED       = 1,  /* Filter cannot be modified */
      NPF_GCLASS_FILTER_CONFIGURABLE= 2,  /* Rules may be changed */
      NPF_GCLASS_FILTER_CLONABLE    = 3,  /* Filter can be cloned */
} NPF_gclassFilterAttr_t;
```

4.1.5 Protocol Type

The NPF Protocol Type structure (**`NPF_protocol_t`**) was published in the NPF Packet Handler API Implementation Agreement, Revision 1.0 (September, 2003). The definition here extends the Protocol Family enumeration to include NPF metadata types; this definition supercedes the previously published one.

The representation of metadata as “protocols” is done as a convenience for the definition of filters. This does not mean that metadata is a protocol, or that the implementation must represent metadata as a protocol header of some kind.

```
/************************************
  Protocol Type Indicator

  This structure identifies a protocol by its protocol number.
  Protocol numbers are assigned by several different numbering
  authorities, so there is a protocol family identifier to say
  what number series the protocol number belongs to.  The IEEE
  and IETF number assignments can be found on the IANA web
  site,http://www.iana.org/numbers.html.  The “NPF Metadata”
  family identifies a number of metadata types; these type
  codes are used in the “protocol” variable of NPF_protocol_t.
************************************/

typedef enum {
      NPF_PROTO_FAM_LLC = 1,  /* IEEE 802 LSAP assignments */
      NPF_PROTO_FAM_ET  = 2,  /* IEEE EtherType assignments */
      NPF_PROTO_FAM_PPP = 3,  /* IETF PPP protocol numbers */
      NPF_PROTO_FAM_IP  = 4,  /* IETF IP protocol numbers */
      NPF_PROTO_FAM_NPF = 5,  /* NP Forum-defined numbers */
      NPF_PROTO_FAM_UNK = 6,  /* Unknown protocol */
      NPF_PROTO_FAM_ANY = 7,  /* Wildcard value */
      NPF_PROTO_FAM_META = 8  /* NPF Metadata */
} NPF_protocolFamily_t;

typedef struct {
      NPF_protocolFamily_t    family;
      NPF_uint16_t            protocol;
} NPF_protocol_t;
```

4.1.6 Field Definition

The following structure specifies a “field” that is examined by a filter. A field consists of a contiguous string of bits within a PDU, specified by protocol type, bit offset and length.

```
typedef struct    {
    NPF_protocol_t      protocol;   /* Protocol or metadata type */
    NPF_boolean_t       validShID;  /* TRUE if subHeaderID is significant */
    NPF_uint32_t        subHeaderID;/* Subheader type code (e.g. option #) */
    NPF_uint32_t        offset;     /* starting bit offset within header */
    NPF_uint32_t        length;     /* length of the field in bits */
} NPF_gclassField_t;
```

Some protocols have variable-length fields that cause the fields following it not to lie in fixed positions within the PDU; their positions can only be found by examining the preceding fields. For fields following a variable-length field in a PDU, the sub-header ID identifies a sub-header within the PDU, so that its subfields can be referenced by a fixed offset.
For example, some protocols have headers that contain type-length-value (TLV) encoded options. An individual option can be identified using its type code as the **`subHeaderID`**, and the protocol number of the protocol that defines the type code.
The **`validShID`** field is a boolean value that indicates whether the **`subHeaderID`** field contains any information. If TRUE, the **`subHeaderID`** identifies the relative starting point for the offset value; if FALSE, the offset is from the beginning of the PDU.

We assume that the underlying implementation of a filter has the means to locate metadata and the start of the PDU for any protocol ID. In some implementations this is known from the filter's context: control is always passed to it with a pointer to the start of a particular PDU it is supposed to examine. If a multi-field filter examines fields of more than one protocol type, the underlying implementation is assumed to know how to locate each PDU within the packet.

4.1.7 Filter Definition

The filter definition structure proper points to an array of field definitions. Each field has a number, according to the order in which field definitions are listed: field 0 is the first one listed; field 1 is the second, and so on. Each filter has an optional name, an ascii string of up to 16 characters. The filter definition may list fields in any order: not necessarily the order in which fields appear in the packet.

```
typedef struct    {
    NPF_FilterID_t      filterNum;  /* FilterID of this filter */
    NPF_gclassFilterType_t type;    /* Precedence algorithm type */
    NPF_uint32_t        nfields;    /* Number of fields examined */
    NPF_gclassField_t   *field;     /* Array of field definitions */
    NPF_uint32_t        nRules;     /* Number of active rules */
    NPF_gclassFilterAttr_t attr;    /* Filter attribute */
} NPF_gclassFilter_t;
```

4.1.8 Patterns

A pattern is a value or a range of values (possibly with "don't-care" bits). The value or range is compared with one of the fields examined by a filter.

`fieldNum`: For a multifield filter, this identifies which field the value is compared with. If the filter examines only one field, this value is not used. Zero refers to the 1st field in the array, 1 refers to the 2nd, and so on.

`fieldLen`: This is included for convenience. It must be identical to the field length corresponding to **`fieldNum`** in the filter to which the pattern is assigned.

`type:` A pattern is either of type “value,” “range,” “mask,” or “prefix.” A “value” pattern matches a single value in a protocol field. A “range” pattern matches a range of values in the field; any number in the field, lying between the first and second values, inclusive, is considered a match. A “mask” type pattern uses a single value and a bit-mask. Any field pattern whose bits match those of the value in positions where the corresponding mask bit is *set* is considered a match. A “prefix” type pattern consists of a value and a prefix length ***n***; a field matches if it contains the same value in its first ***n*** bits. Prefix and mask are both ways of indicating “don’t-care” bits. Whereas a prefix limits the don’t-cares to a contiguous field extending to the least significant bit of the value, a mask can place don’t-care bits anywhere in the value, with no requirement that they be contiguous. ***LPM and No Precedence filters can use patterns of type “prefix,” “range,” and “value” only. Ordered Precedence filters can use all types; “mask” patterns are valid only with Ordered Precedence filters.***

`value:` A value is contained in an array of one or more bytes. The length of this array MUST be equal to or greater than the length of the field in bytes, rounded up. If **`prefixLen`** is zero, **`*value`** MUST be a null pointer (zero). A pattern value is stored with its first bit in the most significant bit position of the first byte of the **`value`** array. Values longer than eight bits are stored in Network Byte Order, meaning, for example, that a 32-bit IPv4 address is stored with its most significant 8 bits in the first byte, and so on, ending with the least significant eight bits in the fourth byte. If the pattern type is a range of values, the second value is used; this must be greater than the first value.

`prefixLen:` The number of significant bits in the **`value`**.

`mask`: Used only if the pattern type is “mask,” this is a bit-string stored according to the same rules as **`value`** above.

```
typedef enum      {
      NPF_GCLASS_PATTERN_TYPE_VALUE = 1,
      NPF_GCLASS_PATTERN_TYPE_RANGE = 2,
      NPF_GCLASS_PATTERN_TYPE_MASK = 3,
      NPF_GCLASS_PATTERN_TYPE_PREFIX = 4
} NPF_gclassPatternType_t;

typedef struct {
    NPF_uint32_t  fieldNum;   /* Field number within the filter */
    NPF_uint32_t  fieldLen;   /* Field length in bits */
    NPF_gclassPattType_t type;
    union {
          struct {
                NPF_uint8_t *value;
          } value;
          struct {
                NPF_uint8_t *minValue;
                NPF_uint8_t *maxValue;
          } range;
          struct {
                NPF_uint8_t *value;
                NPF_uint8_t *mask;
          } mask;
          struct {
                NPF_uint8_t *value;
                NPF_uint32_t  prefixLen;
          } prefix;
     } u;
} NPF_gclassPatt_t;
```

Patterns have different constraints according to the class and type of filter they are used in. Table 1 summarizes them.

	No Precedence	**Ordered Precedence**	**LPM**
Single Field	- Value type only - no don't-care bits	- Value, Range, Prefix or Mask types - each rule has a position index.	- Range or Prefix types - zero or more don't-care bits.
Multi-Field	- Multiple Value types - no don't-care bits	- Multiple Value, Range, Prefix or Mask types - each rule has a position index.	Does not apply

Table 1: Pattern characteristics by filter class and type

Note: Although a "Prefix" type pattern is allowed in Ordered Precedence filters, this is only for convenience in delimiting the don't-care bits; it does NOT imply LPM operation within the filter.

4.1.9 Action Types

Each pattern has an associated action. Actions are of two types:

- Return metadata value
- Go to filter

```
typedef enum {
    NPF_filActTypeClassID = 1,      /* Return a ClassID value */
    NPF_filActTypeNextFilter = 2,         /* Go to next filter, same LFB */
} NPF_gclassActionType_t;

typedef struct {
    NPF_gclassActionType_t type;
    union   {
      NPF_uint32_t      classID;    /* Value to be returned */
      NPF_FilterID_t    nextFilter; /* next filter, same LFB */
    } u;
} NPF_gclassAction_t;
```

4.1.10 ClassID Values

One ClassID value, zero, is reserved for use in default rules, to indicate specifically that a packet matched no other rule in a filter.

```
#define NPF_F_GCLASS_CLASSID_VALUE_NO_MATCH 0
```

4.1.11 Rules

A Rule consists of a pattern, or set of patterns, and an action, assigned to a particular filter. A rule must contain a number of patterns equal to the number of fields in the filter it belongs to. A packet is considered to "match" a rule if it satisfies, along with any associated metadata, *all* the patterns associated with the rule.

```
typedef struct    {
    NPF_uint32_t        numPatts;   /* Number of patterns in the rule */
    NPF_gclassPatt_t    *pattern;   /* Pointer to array of patterns */
    NPF_gclassAction_t  action;     /* Action for this rule */
} NPF_gclassRule_t;
```

With No-precedence and LPM filters, a rule's pattern defines its place implicitly with respect to other rules. For Ordered Precedence filters, more information is needed. The following is the definitions needed for managing Ordered Precedence filters:

```
typedef enum {
    NPF_gclassOP_CmdInsert = 1,     /* Insert a rule */
    NPF_gclassOP_CmdDelete = 2,     /* Delete one or more rules */
    NPF_gclassOP_CmdReplace = 3,    /* Replace one or more rules */
} NPF_gclassOP_Cmd_t;

typedef struct    {
    NPF_uint32_t        ruleNumber; /* ord. of rule inserted or replacing */
    NPF_gclassOP_Cmd_t  command;    /* Insert, Delete, or Replace */
    NPF_uint32_t        from;       /* Lower ordinal of affected range */
    NPF_uint32_t        to;         /* Upper ordinal of affected range */
    NPF_gclassRule_t    rule;       /* Rule definition structure */
} NPF_gclassOP_Rule_t;
```

The **`ruleNumber`** variable is used in Ordered Precedence filters to indicate where the rule is to be placed in the set. Rule ordinals are positive integers chosen by the application; the lower the ordinal, the higher its precedence. (When a packet could match more than one rule, it always matches the one with the lowest ordinal.) It is not possible to insert a rule between two other rules whose ordinals differ by one; you the application must first renumber the existing rules to create an opening in the sequence, by deleting them and adding them again with different ordinals, or by doing a "replace" operation that replaces the existing rules with renumbered copies.
The **`to`** and **`from`** variables are used with Delete and Replace commands, to indicate the range (inclusive) of existing rules affected. They are ignored on Insert commands. The Insert command replaces a single existing rule if it has the same ordinal as the new rule being inserted. The **`rule`** and **`ruleNumber`** variables are ignored on a Delete command.

We also need structures that comprise a set of rules; these structures are used in callbacks from rule query functions.

```
typedef struct    {
      NPF_uint32_t          nRules;     /* Number of rules in the array */
      NPF_gclassRule_t      *rules;     /* Pointer to array of rules */
} NPF_gclassRuleSet_t;
```

```
typedef struct    {
      NPF_uint32_t            nRules;     /* Number of rules in the array */
      NPF_gclassOP_Rule_t     *rules;     /* Pointer to array of rules */
} NPF_gclassOP_RuleSet_t;
```

4.1.11.1 Default Rule

A Default Rule is a special usage of the **`NPF_gclassRule_t`** structure in which the **`numPatts`** field is zero. When **`numPatts`** is zero, the **`pattern`** field should be encoded as a null value; the implementation MUST ignore it. The **`action`** field indicates the filter's action for any packet that matches no other rule. A filter can have at most one default rule.

N.B. in LPM filters, a rule containing a pattern with zero for the **`prefixLen`** value explicitly matches all packets. Such a rule in an LPM filter SHALL match packets in preference to the Default Rule.

An implementation MUST incorporate a built-in default rule in each filter. This rule MUST incorporate the action "return metadata value zero."

4.2 Data Structures for Completion Callbacks

A completion callback is defined for each of the functions in this API (not included are the callback and event registration/deregistration functions, which are synchronous). The callback response contains one of the following codes, indicating the function that was called to cause the callback. This code tells the application how to interpret the data included in the union that is part of the response structure.

```
/*
 * This structure contains an array of filter IDs,
 * along with the array length.  It is used in the response
 * from NPF_F_gclassLFB_AttrGet().
 */
typedef struct    {
      NPF_uint32_t      nFilters;   /* Number of filters in LFB */
      NPF_FilterID_t    *filters;   /* One for each filter */
} NPF_gclassFilterList_t;

/*
 * LFB Attribute Structure
 */
typedef struct    {
      NPF_boolean_t     canCreate;  /* TRUE if new filters can be created */
      NPF_boolean_t     canClone;   /* TRUE if filters can be cloned */
      NPF_F_Metadata_Id_t metaTag;  /* The type of metadata output */
      NPF_gclassFilterList_t filterList;  /* List of active filters */
} NPF_gclassLFB_Attr_t;
```

4.2.1 Asynchronous Response

```
/*
 * An asynchronous response contains an error/success code,
 * other optional information that correlates the response
 * to an element in a request array, and in some cases a
 * function-specific structure embedded in a union. One or
 * more of these is passed to the callback function as an
 * array within the NPF_gclassCallbackData_t structure.
 */
typedef struct {  /* Asynchronous Response Structure */
      NPF_gclassReturnCode_t  error;       /* Error code for this response */
      union {           /* Function-specific structures: */
        NPF_gclassFilter_t    filterAttr; /* NPF_F_gclassFilterAttrGet() */
        NPF_uint32_t          ruleNum;          /* RuleStore/Set/Del fcns */
        NPF_gclassLFB_Attr_t  lfbAttr;    /* NPF_F_gclassLFB_AttrGet() */
        NPF_uint32_t          nRules;     /* FilterAttrQuery() */
        NPF_gclassRuleSet_t   ruleSet;    /* NPF_F_gclassLPM_RuleGet() */
        NPF_gclassOP_RuleSet_t OP_RuleSet;/* NPF_F_gclassOP_RuleGet() */
        NPF_filterID_t        filterID;   /* NPF_F_gclassFilterClone() */
        NPF_uint32_t          unused;     /* All other functions */
      } u;
} NPF_gclassAsyncResponse_t;
```

4.2.2 Callback Type

```
/*
 * Completion callback types
 */
typedef enum NPF_gclassCallbackType {
      NPF_GCLASS_FILTER_ATTR_GET          = 1,
      NPF_GCLASS_LPM_RULE_STORE           = 2,
      NPF_GCLASS_LPM_RULE_SET_REPLACE     = 3,
      NPF_GCLASS_LPM_RULE_DELETE          = 4,
      NPF_GCLASS_OP_RULE_SET              = 5,
      NPF_GCLASS_RULE_FLUSH               = 6,
      NPF_GCLASS_FILTER_ATTR_QUERY        = 7,
      NPF_GCLASS_FILTER_LIST_GET          = 8,
      NPF_GCLASS_FILTER_CLONE             = 9,
      NPF_GCLASS_LPM_RULE_GET             = 10,
      NPF_GLCASS_OP_RULE_GET              = 11
} NPF_gclassCallbackType_t;
```

4.2.3 Callback Data

```
/*
 * The callback function receives the following structure containing
 * one or more asynchronous responses from a single function call.
 * There are several possibilities:
 * 1. The called function does a single request
 *    - numCallbackResp = 1, and the resp array has just one element.
 *    - allOK = TRUE if the request completed without error
 *      and the only return value is the response code.
 *    - if allOK = FALSE, the "resp" structure has the error code.
 * 2. the called function supports an array of requests
 *    a. All completed successfully, at the same time
 *       - allOK = TRUE, n_resp = 0.
 *    b. Some completed, but not all, or there are values besides
 *       the response code to return:
 *       - allOK = FALSE, n_resp = the number completed
 *       - the "resp" array will contain one element for
 *         each completed request, with the error code
 *         in the NPF_gclassAsyncResponse_t structure, along
 *         with any other information needed to identify
 *         which request element the response belongs to.
 *       - Callback function invocations are repeated in
 *         this fashion until all requests are complete.
 *         Responses are not repeated for request elements
 *         already indicated as complete in earlier callback
 *         function invocations.
 */
typedef struct {
    NPF_gclassCallbackType_t  type; /* Identifies the function called */
    NPF_boolean_t             allOK; /* TRUE if all requests completed OK*/
    NPF_uint32_t              numResp;/* Number of responses in array */
    NPF_gclassAsyncResponse_t *resp; /* Array of response structures */
} NPF_gclassCallbackData_t;
```

Callback Type	Callback Data
NPF_GCLASS_LFB_ATTR_GET	NPF_gclassFilterList_t
NPF_GCLASS_FILTER_ATTR_GET	NPF_gclassFilter_t
NPF_GCLASS_LPM_RULE_STORE	NPF_uint32_t (Rule Number)
NPF_GCLASS_LPM_RULE_SET_REPLACE	NPF_uint32_t (Rule Number)
NPF_GCLASS_LPM_RULE_DELETE	NPF_uint32_t (Rule Number)
NPF_GCLASS_OP_RULE_SET	NPF_uint32_t (Rule Number)
NPF_GCLASS_RULE_FLUSH	Unused
NPF_GCLASS_FILTER_ATTR_QUERY	NPF_uint32_t (number of rules)
NPF_GCLASS_FILTER_CLONE	NPF_filterID_t
NPF_GCLASS_LPM_RULE_GET	NPF_gclassRuleSet_t
NPF_GLCASS_OP_RULE_GET	NPF_gclassOP_RuleSet_t

Table 2. Callback type to Callback data mapping table

4.3 Data Structures for Event Notifications

The Generic Classifier LFB does not define or generate any events.

4.4 Error Codes

4.4.1 Common NPF Error Codes

- **NPF_NO_ERROR --** This value MUST be returned when a function was successfully invoked. This value is also used in completion callbacks where it MUST be the only value used to signify success.
- **NPF_E_UNKNOWN --** An unknown error occurred in the implementation such that there is no error code defined that is more appropriate or informative.
- **NPF_E_BAD_CALLBACK_HANDLE --** A function was invoked with a callback handle that did not correspond to a valid NPF callback handle as

returned by a registration function, or a callback handle was registered with a registration function belonging to a different API than the function call where the handle was passed in.

- **NPF_E_BAD_CALLBACK_FUNCTION --** A callback registration was invoked with a function pointer parameter that was invalid.
- **NPF_E_CALLBACK_ALREADY_REGISTERED --** A callback or event registration was invoked with a pair composed of a function pointer and a user context which was previously used for an identical registration.
- **NPF_E_FUNCTION_NOT_SUPPORTED --** This error value MUST be returned when an optional function call is not implemented by an implementation. This error value MUST NOT be returned by any required function call. This error value MUST be returned as the function return value (i.e. synchronously).
- **NPF_E_RESOURCE_EXISTS --** A duplicate request to create a resource was detected. No new resource was created.
- **NPF_E_RESOURCE_NONEXISTENT --** A duplicate request to destroy or free a resource was detected. The resource was previously destroyed or never existed.

4.4.2 LFB Specific Error Codes

```
/*
 * Asynchronous error codes (returned in function callbacks)
 */
#define GC_ERRNPF_GC_ERR(n) ((NPF_gclassReturnCode_t) NPF_GCLASS_BASE_ERR +
(n))

#define NPF_E_GCLASS_INVALID_FILTER            GC_ERRNPF_GC_ERR(0)
#define NPF_E_GCLASS_NO_SUCH_RULE              GC_ERRNPF_GC_ERR(1)
#define NPF_E_GCLASS_UNKNOWN                   GC_ERRNPF_GC_ERR(2)
#define NPF_E_GCLASS_INVALID_RULE              GC_ERRNPF_GC_ERR(3)
#define NPF_E_GCLASS_INVALID_RULE_NUMBER       GC_ERRNPF_GC_ERR(4)
#define NPF_E_GCLASS_INVALID_PATTERN           GC_ERRNPF_GC_ERR(5)
#define NPF_E_GCLASS_INVALID_ACTION            GC_ERRNPF_GC_ERR(6)
#define NPF_E_GCLASS_INVALID_RANGE             GC_ERRNPF_GC_ERR(7)
#define NPF_E_GCLASS_INVALID_COMMAND           GC_ERRNPF_GC_ERR(8)
#define NPF_E_GCLASS_INVALID_FE_HANDLE         GC_ERRNPF_GC_ERR(9)
#define NPF_E_GCLASS_INVALID_LFB_ID            GC_ERRNPF_GC_ERR(10)
#define NPF_E_GCLASS_OPTION_NOT_SUPPORTED      GC_ERRNPF_GC_ERR(11)
#define NPF_E_GCLASS_FILTER_NOT_CLONABLE       NPF_GC_ERR(12)

typedef NPF_uint32_t NPF_gclassReturnCode_t;
```

5 Functional APIs (FAPIs)

5.1 Required Functional APIs

5.1.1 Completion Callback Function

Syntax

```
typedef void (*NPF_F_gclassCallBackFunc_t)(
 NPF_IN NPF_userContext_t                  userContext,
 NPF_IN NPF_correlator_t                   correlator,
 NPF_IN NPF_F_gclassCallbackData_t   gclassCallbackData);
```

Description

This callback function is used by the application to register an asynchronous response handling routine with the Generic Classifier FAPI implementation. This callback function is intended to be implemented by the application, and registered with the Generic Classifier FAPI implementation through the callback registration function.

Input Parameters

- **`userContext`** – The context item that was supplied by the application when the completion callback function was registered.
- **`correlator`** – The correlator item that was supplied by the application when the FAPI function call was made. The correlator is used by the application mainly to distinguish between multiple invocations of the same function.
- **`gclassCallbackData`** – Response information related to the FAPI function call. Contains information that is common among all functions, as well as information that is specific to a particular function. See the **`NPF_F_gclassCallbackData_t`** definition, section 3.2.3, for details.

Output Parameters

None.

Return Value

None.

Asynchronous Response

Not Applicable.

Notes

None.

5.1.2 Completion Callback Registration Function

Syntax

```
NPF_error_t NPF_F_gclassRegister(
 NPF_IN NPF_userContext_t                  userContext,
 NPF_IN NPF_F_gclassCallbackFunc_t         gclassCallbackFunc,
 NPF_OUT NPF_callbackHandle_t             *gclassCallbackHandle);
```

Description

This function is used by the application to register its completion callback function for receiving asynchronous responses related to NPF FAPI function calls. The application may register multiple callback functions using this function. The callback function is identified by the pair of userContext and gclassCallbackFunc, and for each individual pair, a unique gclassCallbackHandle will be assigned for future reference. Since the callback function is identified by both userContext and gclassCallbackFunc, 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 gclassCallbackFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return an error that indicates that the callback has already been registered.

Note : **`NPF_F_gclassRegister( )`** is a synchronous function and has no completion callback associated with it.

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. Application can assign any value to the userContext and the value is completely opaque to the NPF FAPI implementation.
- **`gclassCallbackFunc`** – The pointer to the completion callback function to be registered.

Output Parameters

- **`gclassCallbackHandle`** – A unique identifier assigned for the registered userContext and gclassCallbackFunc pair. This handle will be used by the application to specify which callback function to be called when invoking asynchronous NPF FAPI functions. It will also be used when de-registering the userContext and gclassCallbackFunc pair.

Return Values

- **NPF_NO_ERROR** – The registration completed successfully.
- **NPF_E_BAD_CALLBACK_FUNCTION** – The callback function is NULL.
- **NPF_E_CALLBACK_ALREADY_REGISTERED** – No new registration was made since the userContext and gclassCallbackFunc pair were already registered.

Note: Whether this should be treated as an error or not is dependent on the application.

Asynchronous Response

Not Applicable.

Notes

None.

5.1.3 Completion Callback Deregistration

Syntax

```
NPF_error_t NPF_F_gclassDeregister(
 NPF_IN NPF_callbackHandle_t   gclassCallbackHandle);
```

Description

This function is used by the 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_gclassDeregister( )`** is a synchronous function and has no completion callback associated with it.

Input Parameters

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

Output Parameters

None.

Return Values

- **NPF_NO_ERROR** – The de-registration completed successfully.
- **NPF_E_BAD_CALLBACK_HANDLE** – The function does not recognize the callback handle. There is no effect to the registered callback functions.

unousAsynchronous Response

Not Applicable.

Notes

None.

5.2 Functional APIs

This section describes the functions defined in the Generic Classification API. These functions are:

- **`NPF_F_gclassLFB_AttrGet()`**: given a Generic Classifier LFB ID, returns the LFB attributes, including a list of filter IDs.
- **`NPF_F_gclassFilterAttrGet()`**: given a filter ID, returns filter attributes.
- **`NPF_F_gclassLPM_RuleStore()`**: adds or replaces one or more rules in an LPM or No Precedence filter.
- **`NPF_F_gclassLPM_RuleSetReplace()`**: adds a set of rules to an LPM or No Precedence filter, replacing any it already has.
- **`NPF_F_gclassLPM_RuleDelete()`**: Deletes one or more rules from an LPM or No Precedence filter.
- **`NPF_F_gclassOP_RuleSet()`**: Performs operations on Ordered Precedence rule sets.
- **`NPF_F_gclassRuleFlush()`**: Removes all rules from a filter.
- **`NPF_F_gclassFilterAttrQuery()`**: Queries a filter's available rule storage capacity.
- **`NPF_F_gclassFilterClone()`**: Creates a new filter by cloning an existing one.
- **`NPF_F_gclassLPM_RuleGet()`**: Retrieves all the active rules from an LPM or No Precedence filter.
- **`NPF_F_gclassOP_RuleGet()`**: Retrieves all the active rules from an Ordered Precedence filter.

5.2.1 NPF_F_gclassLFB_AttrGet()

```
NPF_error_t NPF_F_gclassLFB_AttrGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID);
```

5.2.1.1 Description

Given an LFB's ID, this function returns, in a callback response, an LFB Attribute structure (**`NPF_gclassLFB_Attr_t`**) containing a list of the filters this LFB holds. Each filter is known by a filter ID, consisting of the LFB's ID and a number value assigned by the LFB.

5.2.1.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: ignored; the callback cannot be suppressed. This parameter is included only for consistency.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier logical function block queried.

5.2.1.3 Output Parameters

None.

5.2.1.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.1.5 Callback Response

In the callback, the **`NPF_gclassAsyncResponse_t`** structure contains a **`NPF_gclassLFB_Attr_t`** within the union. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The callback response includes an **`NPF_gclassFilterList_t`** structure indicating how many filters exist, and pointing to an array of the filter IDs defined for the LFB.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID. No filter IDs are returned.

5.2.2 NPF_F_gclassFilterAttrGet()

```
NPF_error_t NPF_F_gclassFilterAttrGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);
```

5.2.2.1 Description

Given a filter's ID, this function returns, in a callback response, an **`NPF_gclassFilter_t`** structure containing the attributes of a filter: type, number of fields, field definitions, and number of rules it contains.

5.2.2.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: ignored; the callback cannot be suppressed. This parameter is included only for consistency.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: ID of the Classifier Logical Function Block.
- **`filter`**: the filter's ID value.

5.2.2.3 Output Parameters

None.

5.2.2.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.2.5 Callback Response

In the callback, the **`NPF_gclassAsyncResponse_t`** structure contains a **`NPF_gclassFilter_t`** within the union. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The callback response includes an **`NPF_gclassFilter_t`** structure containing the attributes of the filter.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID value. No filter attributes are returned.

5.2.3 NPF_F_gclassLPM_RuleStore()

```
NPF_error_t NPF_F_gclassLPM_RuleStore(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nRules,
      NPF_IN NPF_gclassRule_t       *ruleArray);
```

5.2.3.1 Description

This function adds or replaces one or more rules in a LPM or No Precedence filter. If a rule in the **`ruleArray`** has the same pattern set and length as an existing rule, the existing rule is replaced. If a rule in the **`ruleArray`** does not match the pattern setand length in any existing rule, the new rule is added to the set.

5.2.3.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: desired level of callback verbosity: always, never, or only upon error.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block
- **`filter`**: the ID of the filter to be altered.
- **`nRules`**: the number of rules in the **`ruleArray`**.
- **`ruleArray`**: a pointer to an array of rules.

5.2.3.3 Output Parameters

None.

5.2.3.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.3.5 Callback Response

The callback response includes the index (**`NPF_uint32_t  ruleNum`**) of the element of the request array (from 0 to N-1) that succeeded or failed. If callbacks are requested for all results, the total number of response array elements will be equal to the value of **`nRules`** given in the call. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: A request element completed successfully.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.

- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID. No action was performed.
- **`NPF_E_GCLASS_INVALID_RULE`**: Something about the rule was not acceptable to the implementation.
- **`NPF_E_GCLASS_INVALID_PATTERN`**: Something about the pattern was not acceptable to the implementation, or the pattern type was not valid for an LPM filter.
- **`NPF_E_GCLASS_INVALID_ACTION`**: Something about the action was not acceptable to the implementation.

5.2.4 NPF_F_gclassLPM_RuleSetReplace()

```
NPF_error_t NPF_F_gclassLPM_RuleSetReplace(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nRules,
      NPF_IN NPF_gclassRule_t       *ruleArray);
```

5.2.4.1 Description

This function replaces all the rules in a LPM or No Precedence filter with a new set of rules. Although this is equivalent to calling **`NPF_F_gclassLPM_RuleFlush()`** followed by **`NPF_F_gclassLPM_RuleStore()`**, this function is provided so that the operation can be made to occur atomically – as might be needed in some cases where filters provide security against attacks or unauthorized access.

5.2.4.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: desired level of callback verbosity: always, never, or only upon error.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID:`** the ID of the Classifier Logical Function Block.
- **`filter`**: the ID value of the filter to be altered.
- **`nRules`**: the number of rules in the **`ruleArray`**.
- **`ruleArray`**: a pointer to an array of rules.

5.2.4.3 Output Parameters

None.

5.2.4.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.4.5 Callback Response

The callback response includes the index (**`NPF_uint32_t  ruleNum`**) of the element of the request array (from 0 to N-1) that succeeded or failed. If callbacks are requested for all results, the total number of response array elements will be equal to the value of **`nRules`** given in the call. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: A request element completed successfully.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.

- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID. No rules were added or modified.
- **`NPF_E_GCLASS_INVALID_RULE`**: Something about the rule was not acceptable to the implementation.
- **`NPF_E_GCLASS_INVALID_PATTERN`**: Something about the pattern was not acceptable to the implementation, or the pattern type was not valid for an LPM filter..
- **`NPF_E_GCLASS_INVALID_ACTION`**: Something about the action was not acceptable to the implementation.

5.2.5 NPF_F_gclassLPM_RuleDelete()

```
NPF_error_t NPF_F_gclassLPM_RuleDelete(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nPatternsnRules,
      NPF_IN NPF_gclassPattgclassRule_t         *pattArrayruleArray);
```

5.2.5.1 Description

This function deletes one or more rules in a LPM or No Precedence filter. If a pattern or set of patterns in the **pattArray ruleArray** matches an existing rule, the existing rule is deleted (all patterns must be the same). If a pattern in the **pattArray ruleArray** does not match the patterns and length in any existing rule, an error is returned.

5.2.5.2 Input Parameters

- **cbHandle**: the callback handle returned by **NPF_F_gclassRegister()**.
- **correlator**: a 32-bit value that will be returned in the callback for this function call.
- **cbDesired**: desired level of callback verbosity: always, never, or only upon error.
- **feHandle**: the handle of the Forwarding Element in which the LFB is located.
- **lfbID:** the ID of the Classifier Logical Function Block.
- **filter**: the ID of the filter to be altered.
- **PatternsnRules**: the number of rules in the **pattArrayruleArray**.
- **pattArrayruleArray**: a pointer to an array of patternsrules. The **action** field in these rules is ignored, and may be given as zero.

5.2.5.3 Output Parameters

None.

5.2.5.4 Immediate Return Codes

- **NPF_NO_ERROR**: function call accepted, and a callback will occur or has already occurred.
- **NPF_E_BAD_CALLBACK_HANDLE**: the **cbHandle** parameter is invalid; no callback will occur.

5.2.5.5 Callback Response

The callback response includes the index (**NPF_uint32_t ruleNum**) of the element of the request array (from 0 to N-1) that succeeded or failed. The callback can return one of the following response codes:

- **NPF_NO_ERROR**: A rule was deleted successfully.
- **NPF_E_GCLASS_INVALID_FE_HANDLE**: The **feHandle** parameter is not a valid FE handle.

- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_INVALID_PATTERN`**: The given rule contained a pattern that was not of a valid type for LPM filters.
- **`NPF_E_GCLASS_NO_SUCH_RULE`**: A rule was not deleted because it was not found in the existing rule set.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID. No rules were deleted.

5.2.6 NPF_F_gclassOP_RuleSet()

```
NPF_error_t NPF_F_gclassOP_RuleSet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nRules,
      NPF_IN NPF_gclassOP_Rule_t    *ruleArray);
```

5.2.6.1 Description

This function performs insert, delete, and replace operations on rules in an Ordered Precedence filter. Each rule in the ruleArray contains its own command value to indicate what is to be done with that rule. Possible commands are:

- **`NPF_gclassOP_CmdInsert`**: The rule is added to the filter. If its ordinal matches an existing rule, it replaces that rule. Otherwise, it is inserted in sequence.
- **`NPF_gclassOP_CmdDelete`**: The **`to`** and **`from`** variables in the rule indicate the range of rules to delete. The **`ruleNumber`** and **`rule`** variables in the **`OP_Rule`** structure are ignored.
- **`NPF_gclassOP_CmdReplace`**: The **`to`** and **`from`** variables in the rule indicate the range (inclusive) of rules to replace with the new rule. The existing range is deleted, and the new rule is inserted using the same behavior as the **`NPF_gclassOP_CmdInsert`** command.

5.2.6.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: desired level of callback verbosity: always, never, or only upon error.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block.
- **`filter`**: the ID of the filter to be altered.
- **`nRules`**: the number of rules in the **`ruleArray`**.
- **`ruleArray`**: a pointer to an array of Ordered Precedence rules.

5.2.6.3 Output Parameters

None.

5.2.6.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.6.5 Callback Response

The callback response includes the index (`NPF_uint32_t  ruleNum`) of the element of the request array (from 0 to N-1) that succeeded or failed. The callback can return one of the following response codes:

- `NPF_NO_ERROR`: A request element completed successfully.
- `NPF_E_GCLASS_INVALID_FE_HANDLE`: The `feHandle` parameter is not a valid FE handle.
- `NPF_E_GCLASS_INVALID_LFB_ID`: The `lfbID` parameter was not a valid Classifier LFB ID.
- `NPF_E_GCLASS_UNKNOWN`: an unknown or unspecified error occurred.
- `NPF_E_GCLASS_INVALID_FILTER`: The `filter` parameter was not a valid filter ID. No action was performed.
- `NPF_E_GCLASS_INVALID_RULE`: Something about the rule was not acceptable to the implementation.
- `NPF_E_GCLASS_INVALID_PATTERN`: Something about the pattern was not acceptable to the implementation.
- `NPF_E_GCLASS_INVALID_ACTION`: Something about the action was not acceptable to the implementation.
- `NPF_E_GCLASS_INVALID_RULE_NUMBER`: The rule number was unacceptable to the implementation.
- `NPF_E_GCLASS_INVALID_COMMAND`: The command code in the rule was invalid.
- `NPF_E_GCLASS_INVALID_RANGE`: The `to`/`from` range in the rule was invalid.

5.2.7 NPF_F_gclassRuleFlush()

```
NPF_error_t NPF_F_gclassRuleFlush(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);
```

5.2.7.1 Description

This function removes all the rules of a filter.

5.2.7.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: desired level of callback verbosity: always, never, or only upon error.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block.
- **`filter`**: the ID of the filter to be flushed.

5.2.7.3 Output Parameters

None.

5.2.7.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.7.5 Callback Response

The union within the **`NPF_gclassAsyncResponse_t`** structure in the callback is unused. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The request completed successfully.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID. No action is performed.

5.2.8 NPF_F_gclassFilterAttrQuery()

```
NPF_error_t NPF_F_gclassFilterAttrQuery(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);
```

5.2.8.1 Description

This function returns, in a callback, the approximate amount of free space for new rules for a given filter. This is returned as an integer, indicating the number of rules that may be added in future calls.

This is an optional function.

Note: implementations should SHOULD be conservative in the value they return for this function. In other words, the value should be the amount of free rule space under worst-case conditions, so that the application can be assured that adding this many new rules will succeed.

5.2.8.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: ignored; the callback cannot be suppressed. This parameter is included only for consistency.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block.
- **`filter`**: the ID of the filter to be queried.

5.2.8.3 Output Parameters

None.

5.2.8.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.8.5 Callback Response

The union within the **`NPF_gclassAsyncResponse_t`** structure in the callback contains **`NPF_uint32_t  nRules`**. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The request completed successfully. The callback data structure contains the estimated amount of free space for new rules in this filter, given as an integer number of rules that may be added.

- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred. The callback response does not contain any information about free space.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID. No other information is returned.

5.2.9 NPF_F_gclassFilterClone()

```
NPF_error_t NPF_F_gclassFilterClone(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);
```

5.2.9.1 Description

This function creates a new filter by cloning an existing filter and returns, in a callback, the ID of the new filter created. The new filter has attributes identical to the original, but the rule set is empty except for a default rule (if applicable). This operation is not allowed unless the orignal filter has the **`NPF_GCLASS_FILTER_CLONABLE`** attribute.

This is an optional function.

5.2.9.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: ignored; the callback cannot be suppressed. This parameter is included only for consistency.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block.
- **`filter`**: the ID of the filter to be cloned.

5.2.9.3 Output Parameters

None.

5.2.9.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.
- **`NPF_E_FUNCTION_NOT_SUPPORTED`**: The implementation does not support this function.

5.2.9.5 Callback Response

The union within the **`NPF_gclassAsyncResponse_t`** structure in the callback contains the ID of the new filter, as **`NPF_filterID_t  filterID`**. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The request completed successfully. The callback data structure contains the estimated amount of free space for new rules in this filter, given as an integer number of rules that may be added.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred. The callback response does not contain any information about free space.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid filter ID. No other information is returned.
- **`NPF_E_GCLASS_FILTER_NOT_CLONABLE`**: The specified filter is not clonable.

5.2.10 NPF_F_gclassLPM_RuleGet()

```
NPF_error_t NPF_F_gclassLPM_RuleGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);
```

5.2.10.1 Description

This function returns, in a callback, the complete set of rules for a given LPM or No Precedence filter, as an **`NPF_gclassRuleSet_t`** structure.

This is an optional function.

5.2.10.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: ignored; the callback cannot be suppressed. This parameter is included only for consistency.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block.
- **`filter`**: the ID of the LPM filter to be queried.

5.2.10.3 Output Parameters

None.

5.2.10.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.10.5 Callback Response

If successful, the callback returns a **`NPF_gclassRuleSet_t`** structure that points to an array of rules, one element for each active rule in the filter. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The request completed successfully. The callback data structure contains a pointer to the set of active the estimated amount of free space for new rules in this filter.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred. The callback response does not contain any information about free space.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The **`filter`** parameter was not a valid LPM or No Precedence filter ID. No other information is returned.

5.2.11 NPF_F_gclassOP_RuleGet()

```
NPF_error_t NPF_F_gclassOP_RuleGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           to,
      NPF_IN NPF_uint32_t           from);
```

5.2.11.1 Description

This function returns, in a callback, the complete set of rules for a given Ordered Precedence filter, as an **`NPF_gclassOP_RuleSet_t`** structure.

This is an optional function.

5.2.11.2 Input Parameters

- **`cbHandle`**: the callback handle returned by **`NPF_F_gclassRegister()`**.
- **`correlator`**: a 32-bit value that will be returned in the callback for this function call.
- **`cbDesired`**: ignored; the callback cannot be suppressed. This parameter is included only for consistency.
- **`feHandle`**: the handle of the Forwarding Element in which the LFB is located.
- **`lfbID`**: the ID of the Classifier Logical Function Block.
- **`filter`**: the ID of the Ordered Precedence filter to be queried.
- **`to, from`**: the range of rule numbers to be returned. The function will return all rules with a number greater than or equal to **`to`**, and less than or equal to **`from`**. If **`to`** and **`from`** are both zero, all active rules will be returned.

5.2.11.3 Output Parameters

None.

5.2.11.4 Immediate Return Codes

- **`NPF_NO_ERROR`**: function call accepted, and a callback will occur or has already occurred.
- **`NPF_E_BAD_CALLBACK_HANDLE`**: the **`cbHandle`** parameter is invalid; no callback will occur.

5.2.11.5 Callback Response

If successful, the callback returns a **`NPF_gclassOP_RuleSet_t`** structure that points to an array of rules, one element for each active rule returned. The callback can return one of the following response codes:

- **`NPF_NO_ERROR`**: The request completed successfully. The callback data structure contains the estimated amount of free space for newa pointer to the set of active rules in this filter.
- **`NPF_E_GCLASS_INVALID_FE_HANDLE`**: The **`feHandle`** parameter is not a valid FE handle.
- **`NPF_E_GCLASS_INVALID_LFB_ID`**: The **`lfbID`** parameter was not a valid Classifier LFB ID.
- **`NPF_E_GCLASS_UNKNOWN`**: an unknown or unspecified error occurred. The callback response does not contain any information about free space.
- **`NPF_E_GCLASS_INVALID_FILTER`**: The `filter` parameter was not a valid Ordered Precedence filter ID. No other information is returned.

6 References

[FORCESREQ]	”Requirements for Separation of IP Control and Forwarding”, H. Khosravi, T. Anderson et al, November, 2003 (RFC 3654)
[FAPITOPO]	”FAPI Topology Manager API”, work in progress, Network Processing Forum SWAPI Functional API TG, 2004.
[SWAPICON]	“Software API Conventions Revision 2”, http://www.npforum.org/techinfo/APIConventions2_IA.pdf, Network Processing Forum SWAPI Foundations TG, September 2003

APPENDIX A HEADER FILE INFORMATION

```
/*
 * npf_f_gclass.h
 *
 * Header file for the NPF Generic Classification Functional API
 */
#ifndef NPF_F_GCLASS__H
#define NPF_F_GCLASS__H

/*
 * LFB Type assignment
 */
#define NPF_LFB_TYPE_GENERIC_CLASSIFIER 13

/*
 * Filter ID (a unique attribute of every filter within the LFB)
 */
typedef NPF_uint32_t     NPF_FilterID_t;

/*
 * Filter Type
 */
typedef enum {
      NPF_GCLASS_TYPE_NO_PRECEDENCE = 1,
      NPF_GCLASS_TYPE_ORDERED_PRECEDENCE = 2,
      NPF_GCLASS_TYPE_LPM = 3
} NPF_gclassFilterType_t;

/************************************
  Protocol Type Indicator

  This structure identifies a protocol by its protocol number.
  Protocol numbers are assigned by several different numbering
  authorities, so there is a protocol family identifier to say
  what number series the protocol number belongs to.  The IEEE
  and IETF number assignments can be found on the IANA web
  site,http://www.iana.org/numbers.html.  The “NPF Metadata”
  family identifies a number of metadata types; these type
  codes are used in the “protocol” variable of NPF_protocol_t.
************************************/

typedef enum {
      NPF_PROTO_FAM_LLC = 1,  /* IEEE 802 LSAP assignments */
      NPF_PROTO_FAM_ET  = 2,  /* IEEE EtherType assignments */
      NPF_PROTO_FAM_PPP = 3,  /* IETF PPP protocol numbers */
      NPF_PROTO_FAM_IP  = 4,  /* IETF IP protocol numbers */
      NPF_PROTO_FAM_NPF = 5,  /* NP Forum-defined numbers */
      NPF_PROTO_FAM_UNK = 6,  /* Unknown protocol */
      NPF_PROTO_FAM_ANY = 7,  /* Wildcard value */
      NPF_PROTO_FAM_META = 8  /* NPF Metadata */
} NPF_protocolFamily_t;

typedef struct {
      NPF_protocolFamily_t    family;
      NPF_uint16_t            protocol;
```

```
} NPF_protocol_t;

/*
 * Filter Attribute
 */
typedef enum      {
      NPF_GCLASS_FILTER_FIXED       = 1,  /* Filter cannot be modified */
      NPF_GCLASS_FILTER_CONFIGURABLE= 3,  /* Rules may be changed */
      NPF_GCLASS_FILTER_CLONABLE    = 2,  /* Filter can be cloned */
} NPF_gclassFilterAttr_t;

/*
 * The definition of a Field within a filter
 */
typedef struct    {
    NPF_protocol_t      protocol;   /* IDs the protocol header */
    NPF_boolean_t       validShID;  /* TRUE if subHeaderID is significant */
    NPF_uint32_t        subHeaderID;/* Subheader type code (e.g. option #) */
    NPF_uint32_t        offset;     /* starting bit offset within header */
    NPF_uint32_t        length;     /* length of the field in bits */
} NPF_gclassField_t;

/*
 * Filter description structure
 */
typedef struct    {
    NPF_FilterID_t      filterNum;  /* ID of this filter */
    NPF_gclassFilterType_t type;    /* Precedence algorithm type */
    NPF_uint32_t        nfields;    /* Number of fields examined */
    NPF_gclassField_t   *field;     /* Array of field definitions */
    NPF_uint32_t        nRules;     /* Number of active rules */
    NPF_gclassFilterAttr_t attr;    /* Filter attribute */
} NPF_gclassFilter_t;


/*
 * Pattern Types
 */
typedef enum      {
      NPF_GCLASS_PATTERN_TYPE_VALUE = 1,
      NPF_GCLASS_PATTERN_TYPE_RANGE = 2,
      NPF_GCLASS_PATTERN_TYPE_MASK = 3,
      NPF_GCLASS_PATTERN_TYPE_PREFIX = 4
} NPF_gclassPatternType_t;

/*
 * Pattern Structure
 */

typedef struct {
    NPF_uint32_t  fieldNum;   /* Field number within the filter */
    NPF_uint32_t  fieldLen;   /* Field length in bits */
    NPF_gclassPattType_t type;
    union {
          struct {
                NPF_uint8_t *value;
          } value;
          struct {
                NPF_uint8_t *minValue;
                NPF_uint8_t *maxValue;
          } range;
          struct {
                NPF_uint8_t *value;
                NPF_uint8_t *mask;
          } mask;
          struct {
                NPF_uint8_t *value;
                NPF_uint32_t  prefixLen;
          } prefix;
     } u;
} NPF_gclassPatt_t;


/*
 * Action Structure
 */
typedef enum {
    NPF_filActTypeClassID = 1,      /* Return a value */
    NPF_filActTypeNextFilter = 2,          /* Go to next filter, same LFB */
} NPF_gclassActionType_t;

typedef struct {
    NPF_gclassActionType_t type;
    union   {
      NPF_uint32_t      classID;    /* Value to be returned */
      NPF_FilterID_t    nextFilter; /* next filter, same LFB */
    } u;
} NPF_gclassAction_t;

#define NPF_F_GCLASS_CLASSID_VALUE_NO_MATCH 0

/*
 * Rule Definition Structures
 */
typedef struct    {
    NPF_uint32_t        numPatts;   /* Number of patterns in the rule */
    NPF_gclassPatt_t    *pattern;   /* Pointer to array of patterns */
    NPF_gclassAction_t  action;     /* Action for this rule */
} NPF_gclassRule_t;

typedef enum {
    NPF_gclassOP_CmdInsert = 1,     /* Insert a rule */
    NPF_gclassOP_CmdDelete = 2,     /* Delete one or more rules */
    NPF_gclassOP_CmdReplace = 3,    /* Replace one or more rules */
} NPF_gclassOP_Cmd_t;

typedef struct    {
    NPF_uint32_t        ruleNumber; /* ord. of rule inserted or replacing */
    NPF_gclassOP_Cmd_t  command;    /* Insert, Delete, or Replace */
    NPF_uint32_t        from;       /* Lower ordinal of affected range */
    NPF_uint32_t        to;         /* Upper ordinal of affected range */
    NPF_gclassRule_t    rule;       /* Rule definition structure */
} NPF_gclassOP_Rule_t;
```

```
/*
 * Rule list structures returned from xxxRuleGet() functions
 */
typedef struct    {
      NPF_uint32_t            nRules;     /* Number of rules in the array */
      NPF_gclassRule_t        *rules;     /* Pointer to array of rules */
} NPF_gclassRuleSet_t;

typedef struct    {
      NPF_uint32_t            nRules;     /* Number of rules in the array */
      NPF_gclassOP_Rule_t     *rules;     /* Pointer to array of rules */
} NPF_gclassOP_RuleSet_t;

/*
 * This structure contains an array of filter numbers,
 * along with the array length.  It is used in the response
 * from NPF_F_gclassLFB_AttrGet().
 */
typedef struct    {
      NPF_uint32_t      nFilters;   /* Number of filters in LFB */
      NPF_FilterID_t    *filters;   /* One for each filter */
} NPF_gclassFilterList_t;

/*
 * LFB Attribute Structure
 */
typedef struct    {
      NPF_boolean_t     canCreate;  /* TRUE if new filters can be created */
      NPF_boolean_t     canClone;   /* TRUE if filters can be cloned */
      NPF_gclassFilterList_t filterList;  /* List of active filters */
} NPF_gclassLFB_Attr_t;

/*
 * Asynchronous error codes (returned in function callbacks)
 */
#define GC_ERRNPF_GC_ERR(n) ((NPF_gclassReturnCode_t) NPF_GCLASS_BASE_ERR +
(n))

#define NPF_E_GCLASS_INVALID_FILTER             GC_ERRNPF_GC_ERR(0)
#define NPF_E_GCLASS_NO_SUCH_RULE               GC_ERRNPF_GC_ERR(1)
#define NPF_E_GCLASS_UNKNOWN                    GC_ERRNPF_GC_ERR(2)
#define NPF_E_GCLASS_INVALID_RULE               GC_ERRNPF_GC_ERR(3)
#define NPF_E_GCLASS_INVALID_RULE_NUMBER        GC_ERRNPF_GC_ERR(4)
#define NPF_E_GCLASS_INVALID_PATTERN            GC_ERRNPF_GC_ERR(5)
#define NPF_E_GCLASS_INVALID_ACTION             GC_ERRNPF_GC_ERR(6)
#define NPF_E_GCLASS_INVALID_RANGE              GC_ERRNPF_GC_ERR(7)
#define NPF_E_GCLASS_INVALID_COMMAND            GC_ERRNPF_GC_ERR(8)
#define NPF_E_GCLASS_INVALID_FE_HANDLE          GC_ERRNPF_GC_ERR(9)
#define NPF_E_GCLASS_INVALID_LFB_ID             GC_ERRNPF_GC_ERR(10)
#define NPF_E_GCLASS_OPTION_NOT_SUPPORTED       GC_ERRNPF_GC_ERR(11)
#define NPF_E_GCLASS_FILTER_NOT_CLONABLE        NPF_GC_ERR(12)


typedef NPF_uint32_t NPF_gclassReturnCode_t;

/*
 * Completion callback types
 */
typedef enum NPF_gclassCallbackType {
      NPF_GCLASS_FILTER_ATTR_GET          = 1,
      NPF_GCLASS_LPM_RULE_STORE           = 2,
      NPF_GCLASS_LPM_RULE_SET_REPLACE     = 3,
      NPF_GCLASS_LPM_RULE_DELETE          = 4,
      NPF_GCLASS_OP_RULE_SET              = 5,
      NPF_GCLASS_RULE_FLUSH               = 6,
      NPF_GCLASS_FILTER_ATTR_QUERY        = 7,
      NPF_GCLASS_FILTER_LIST_GET          = 8,
      NPF_GCLASS_FILTER_CLONE             = 9,
      NPF_GCLASS_LPM_RULE_GET             = 10,
      NPF_GLCASS_OP_RULE_GET              = 11
} NPF_gclassCallbackType_t;

/*
 * The callback function receives the following structure containing
 * one or more asynchronous responses from a single function call.
 * There are several possibilities:
 * 1. The called function does a single request
 *    - numCallbackResp = 1, and the resp array has just one element.
 *    - allOK = TRUE if the request completed without error
 *      and the only return value is the response code.
 *    - if allOK = FALSE, the "resp" structure has the error code.
 * 2. the called function supports an array of requests
 *    a. All completed successfully, at the same time
 *       - allOK = TRUE, n_resp = 0.
 *    b. Some completed, but not all, or there are values besides
 *       the response code to return:
 *       - allOK = FALSE, n_resp = the number completed
 *       - the "resp" array will contain one element for
 *         each completed request, with the error code
 *         in the NPF_gclassAsyncResponse_t structure, along
 *         with any other information needed to identify
 *         which request element the response belongs to.
 *       - Callback function invocations are repeated in
 *         this fashion until all requests are complete.
 *         Responses are not repeated for request elements
 *         already indicated as complete in earlier callback
 *         function invocations.
 */
typedef struct {
    NPF_gclassCallbackType_t  type; /* Identifies the function called */
    NPF_boolean_t             allOK; /* TRUE if all requests completed OK*/
    NPF_uint32_t              numResp;/* Number of responses in array */
    NPF_gclassAsyncResponse_t *resp; /* Array of response structures */
} NPF_gclassCallbackData_t;


/*
 * An asynchronous response contains an error/success code,
 * other optional information that correlates the response
 * to an element in a request array, and in some cases a
 * function-specific structure embedded in a union. One or
 * more of these is passed to the callback function as an
 * array within the NPF_gclassCallbackData_t structure.
 */
typedef struct {  /* Asynchronous Response Structure */
      NPF_gclassReturnCode_t  error;       /* Error code for this response */
      union {           /* Function-specific structures: */
        NPF_gclassFilter_t    filterAttr; /* NPF_F_gclassFilterAttrGet() */
        NPF_uint32_t          ruleNum;          /* RuleStore/Set/Del fcns */
        NPF_gclassLFB_Attr_t  lfbAttr;    /* NPF_F_gclassLFB_AttrGet() */
        NPF_uint32_t          nRules;     /* FilterAttrQuery() */
        NPF_gclassRuleSet_t   ruleSet;    /* NPF_F_gclassLPM_RuleGet() */
        NPF_gclassOP_RuleSet_t OP_RuleSet;/* NPF_F_gclassOP_RuleGet() */
        NPF_filterID_t        filterID;   /* NPF_F_gclassFilterClone() */
        NPF_uint32_t          unused;     /* All other functions */
      } u;
} NPF_gclassAsyncResponse_t;

/****************************************************
 *
 *  Function Definitions
 *
 ****************************************************/

/*
 * Client's Completion Callback Function
 */
typedef void (*NPF_F_gclassCallbackFunc_t)(
      NPF_IN NPF_userContext_t            userContext,
      NPF_IN NPF_correlator_t      correlator,
      NPF_IN NPF_gclassCallbackData_t     gclassCallbackData);

/*
 * Completion Callback Registration Function
 */
NPF_error_t NPF_F_gclassRegister(
      NPF_IN NPF_userContext_t            userContext,
      NPF_IN NPF_F_gclassCallbackFunc_t   gclassCallbackFunc,
      NPF_OUT NPF_CallbackHandle_t        *gclassCallbackHandle);

/*
 * Completion Callback Derigistration Function
 */
NPF_error_t NPF_F_gclassDeregister(
      NPF_IN NPF_CallbackHandle_t   callbackHandle);

/*
 * NPF_F_gclassLFB_AttrGet()
 *
 * Retrieve the IDs of filters from the LFB
 */
NPF_error_t NPF_F_gclassLFB_AttrGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN LFB_ID_t               lfbID);

/*
 * NPF_F_gclassFilterAttrGet()
 *
 * Discover a filter's attributes
 */
NPF_error_t NPF_F_gclassFilterAttrGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);

/*
 * NPF_F_gclassLPM_RuleStore()
 *
 * Add or replace rules in an LPM or No Precedence filter.
 */
NPF_error_t NPF_F_gclassLPM_RuleStore(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nRules,
      NPF_IN NPF_gclassRule_t       *ruleArray);

/*
 * NPF_F_gclassLPM_RuleSetReplace()
 *
 * Replace all the rules in an LPM or No Precedence filter
 */
NPF_error_t NPF_F_gclassLPM_RuleSetReplace(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nRules,
      NPF_IN NPF_gclassRule_t       *ruleArray);

/*
 * NPF_F_gclassLPM_RuleDelete()
 *
 * Delete one or more rules in an LPM or No Precedence filter
 */
NPF_error_t NPF_F_gclassLPM_RuleDelete(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nPatternsnRules,
      NPF_IN NPF_gclassPattgclassRule_t         *pattArrayruleArray);

/*
 * NPF_F_gclassOP_RuleSet()
 *
 * Insert/delete/replace rules in an Ordered Precedence filter
 */
NPF_error_t NPF_F_gclassOP_RuleSet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           nRules,
      NPF_IN NPF_gclassOP_Rule_t    *ruleArray);
```

```
/*
 * NPF_F_gclassRuleFlush()
 *
 * Remove all the rules from a filter.
 */
NPF_error_t NPF_F_gclassRuleFlush(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         ilter);

/*
 * NPF_F_gclassFilterAttrQuery()
 *
 * Returns the estimated amount of free space for new rules
 */
NPF_error_t NPF_F_gclassFilterAttrQuery(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);

/*
 * NPF_F_gclassLPM_RuleGet()
 *
 * Retrieve the set of active rules from an LPM or No Precedence filter
 */
NPF_error_t NPF_F_gclassLPM_RuleGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FfilterID_t        filter);

/*
 * NPF_F_gclassOP_RuleGet()
 *
 * Retrieve the set of active rules of an Ordered Precedence filter
 */
NPF_error_t NPF_F_gclassOP_RuleGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           to,
      NPF_IN NPF_uint32_t           from);

NPF_error_t NPF_F_gclassRuleFlush(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);

NPF_error_t NPF_F_gclassFilterAttrQuery(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);


NPF_error_t NPF_F_gclassFilterClone(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);

NPF_error_t NPF_F_gclassLPM_RuleGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter);

NPF_error_t NPF_F_gclassOP_RuleGet(
      NPF_IN NPF_callbackHandle_t   cbHandle,
      NPF_IN NPF_correlator_t       correlator,
      NPF_IN NPF_errorReporting_t   cbDesired,
      NPF_IN NPF_FEHandle_t         feHandle,
      NPF_IN NPF_LFB_ID_t           lfbID,
      NPF_IN NPF_FilterID_t         filter,
      NPF_IN NPF_uint32_t           to,
      NPF_IN NPF_uint32_t           from);

#endif /* NPF_F_GCLASS__H */
```

APPENDIX B ACKNOWLEDGEMENTS

Working Group Chair: Alex Conta

Working Group Editor: John Renwick

Task Group Chair: Alistair Munro

The following individuals are acknowledged for their participation in the FAPI TG teleconferences, plenary meetings, mailing list, and/or for their NPF contributions used for the development of this Implementation Agreement. This list may not be all-inclusive since only names supplied by member companies for inclusion here will be listed. The NPF wishes to thank all active participants to this Implementation Agreement, whether listed here or not.

The list is in alphabetical order of last names:

Gamil Cain, Intel
Philippe Damon, IBM
Alan DeKok, IDT
Jason Goldschmidt, Sun Microsystems
Zsolt Harszti, Ericsson
Todd Kealy, Solidum
Vinoj Kumar, Agere Systems
David Maxwell, IDT
John Renwick, Agere Systems
Michael Speer, Sun Microsystems
Narender Vangati, Agere Systems

APPENDIX C LIST OF COMPANIES BELONGING TO NPF DURING APPROVAL PROCESS

Agere Systems	HCL Technologies	Nortel Networks
Altera	Hifn	NTT Electronics
AMCC	IBM	PMC Sierra
Analog Devices	IDT	Seaway Networks
Avici Systems	Infineon Technologies AG	Sensory Networks
Cypress Semiconductor	Intel	Sun Microsystems
Enigma Semiconductor	IP Fabrics	Teja Technologies
Ericsson	IP Infusion	TranSwitch
Erlang Technologies	Kawasaki LSI	U4EA Group
ETRI	Motorola	Xelerated
EZChip	NetLogic	Xilinx
Flextronics	Nokia	