



Software API Conventions Implementation Agreement

Revision 2.0

Editor(s):

Philippe Damon, damon@fr.ibm.com

Peter Watkins, peter.watkins@cportcorp.com

Copyright © 2003 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 2.0.

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

Table of Contents

1	Revision History.....	4
2	Introduction	5
2.1	Definitions of Normative and Informative	5
2.2	Requirements Language Key Words	5
2.3	Guidance in the use of these Imperatives	6
3	Specification Language	7
4	Naming Conventions.....	8
4.1	Use of the NPF_ prefix	8
4.2	Constants.....	8
4.3	Variables	8
4.4	Type Names	8
4.5	Function Names	8
4.6	Enumerated Values	8
4.7	Abbreviations.....	9
4.8	Function, Type, and Variable Name Lengths	9
4.9	NPF Function Name Composition.....	9
5	Data Types.....	11
5.1	Basic Data Types	11
5.2	Common Data Types	11
5.3	Rules for Construction of New Data Types.....	12
6	Parameter Passing	13
6.1	Scalar Arguments.....	13
6.2	Array Arguments	13
6.3	Resource Handles	13
6.4	Memory Ownership	13
6.5	NPF_IN/NPF_OUT/NPF_IN_OUT Parameters.....	13
6.6	Coherent State Image of Dynamic Elements.....	15
6.7	Support for Local Parameters/Avoidance of Complex Locking	15
6.8	API Signature Guidelines	16
6.9	Packet Buffer Handling	17
7	Function Invocation Model, Events and Completion Callbacks.....	18
7.1	API Completion Callbacks	18
7.2	Lost or Duplicate Callbacks.....	24
7.3	Event notification.....	25
8	Error Handling.....	29
8.1	Synchronous Error Returns.....	29
8.2	Error Code Values	29
9	Compliance and Extensibility	31
9.1	NPF-Defined Optional Functions and Data Structures.....	31
9.2	Revising NPF-Defined APIs.....	31
9.3	Vendor proprietary extensions.....	32
10	Multi-Vendor / Multi-Instance Support	34
11	Design and Implementation Guidelines	35
11.1	Modularity.....	35

11.2 Multicast Invocations..... 35
11.3 Compatibility 36
12 References 37
Appendix A Header File: NPF.h..... 38
Appendix B List of companies belonging to NPF during approval process 40

Table of Figures

[Figure 1 Example strongly typed API function call accepting an array of related items](#) 16
[Figure 2 Example strongly typed API function call returning a set of related items](#) 16
[Figure 3 Generic Control Interface Example](#) 16
[Figure 4 NPF Callback Usage Sequence](#)..... 18
[Figure 5 Multicast Invocation](#) 35

Table of Tables

[Figure 1 Example strongly typed API function call accepting an array of related items](#) 16
[Figure 2 Example strongly typed API function call returning a set of related items](#) 16
[Figure 3 Generic Control Interface Example](#) 16
[Figure 4 NPF Callback Usage Sequence](#)..... 18
[Figure 5 Multicast Invocation](#) 35

1 Revision History

Revision	Date	Reason for Changes
1.0	09/13/2002	Created Rev 1.0 of the implementation agreement by taking the Software Conventions (npf2001.098.28) and making minor editorial corrections.
2.0	09/29/2003	Created Rev 2.0 of the implementation agreement by taking the Software Conventions (npf2003.114.12) and making minor editorial corrections.

2 Introduction

The Network Processor Forum Software API Working group is defining a variety of APIs for the purposes of exposing the functionality of network processors. In order to ensure that the APIs are uniform and consistent in behavior, look, and feel, this document defines a set of conventions that **MUST** be followed by all NPF Software WG API specifications. This document will also define the interoperability goals of the Software API specifications with other NPF and industry specifications.

2.1 Definitions of Normative and Informative

This document defines the following terms for usage here and elsewhere in the Software API Working Group until such time as they have been defined by the NPF operating procedures.

Normative: That portion of a specification that specifies what is required for an implementation to be considered conformant; the mandatory portion of a specification. Note: Specifications may describe non-mandatory (optional) features. Because optional features must satisfy the specification to be considered conformant, their descriptions contain normative text.

Normative information for NPF Software WG specifications **SHALL** only appear in the main text of documents and **MUST NOT** appear in annexes or appendices.

Informative: Portions of a specification document that are included as examples, ancillary information, or that in other ways contribute to common understanding of the specification, but do not specify anything required of an implementation to be considered conformant.

Note that informative information **MAY** be included in either the main text of NPF Software WG documents or in appendices.

2.2 Requirements Language Key Words

In order to enable clear and concise specifications it is necessary to have a uniform set of terminology when describing specifications. This document defines a set of terms that **MUST** be used in all Software API Working Group documents that define any sort of specification or requirement for the behavior of the software of a network processor.

Authors **MUST** incorporate this phrase near the beginning of their document:

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

These key words are defined as follows and **MUST** be capitalized whenever used in a manner intended to specify a behavior or requirement:

1. **MUST** This word, or the terms “**REQUIRED**” or “**SHALL**”, mean that the definition is an absolute requirement of the specification.
2. **MUST NOT** This phrase, or the phrase “**SHALL NOT**”, mean that the definition is an absolute prohibition of the specification.
3. **SHOULD** This word, or the adjective “**RECOMMENDED**”, mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. **SHOULD NOT** This phrase, or the phrase “NOT RECOMMENDED” mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
5. **MAY** This word, or the adjective “OPTIONAL”, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation that does not include a particular option **MUST** be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation that does include a particular option **MUST** be prepared to interoperate with another implementation that does not include the option (except, of course, for the feature the option provides.)

These definitions are an almost verbatim copy of the IETF Best Current Practices Document #14 [2] authored by Scott Bradner. Small changes have been made in order to better align the definitions with NPF Software Working Group needs.

2.3 Guidance in the use of these Imperatives

Imperatives of the type defined in this memo must be used with care and sparingly. In particular, they must only be used where it is actually required for interoperation or to limit behavior that has potential for causing harm (e.g., system crashes). For example, they must not be used to try to impose a particular method on implementors where the method is not required for interoperability.

3 Specification Language

The interface specification language normatively used to define APIs in the Software WG SHALL be C as found in the ANSI C89 Specification [1]. In addition to expressing the APIs in C, informative addendums to the specification SHOULD be made that define an IDL-based representation of the APIs. In defining the APIs, the conventions for constructing APIs listed in the remainder of this document SHALL be followed. These conventions SHOULD be written so as to facilitate a mapping of the APIs to IDL.

4 Naming Conventions

This section will define a set of conventions for naming all parameters, functions, etc., such that their type is easily recognizable across specifications.

4.1 Use of the *NPF_* prefix

The *NPF_* prefix **MUST** be used with functions, constants, types, and parameters that are defined by the NPF. Conversely, the *NPF_* prefix **MUST NOT** be used with functions that are not defined by the NPF. Any implementation of the NPF APIs that uses the prefix *NPF_* in a way that is not found in an NPF specification is considered non-compliant.

4.2 Constants

Constants **MUST** contain only capital letters. Words are separated with underscores and **SHOULD** include scope prefixes:

```
#define NPF_MAX_SCORE 10
```

4.3 Variables

Variable names **MUST** use mixed upper and lower case. The first letter of the first word in a variable name **MUST NOT** be capitalized unless it is an acronym. All other words in a variable name **MUST** start with a capital letter. If the previous word or acronym ends in a capital letter then an underscore **MUST** be inserted: *NPF_daysInWeek*, *NPF_MPLS_Entry*

Some guidelines are:

- Avoid numerals in names: 2-Z, 1-I, 0-O.
- Never use capitalization to differentiate names.
- Homonyms make discussing variables confusing: rap-wrap.

4.4 Type Names

Type names follow the same naming rules as variables with the additional rule that they **MUST** append a *_t* to the end of a type name. This applies to basic types, enumerations, function pointers, and structures.

Example: *NPF_counter_t*, *NPF_someEnumeration_t*, *NPF_someStruct_t*.

4.5 Function Names

Each word in a function name **MUST** start with a capital letter. If the previous word ends in a capital letter then an underscore **MUST** be inserted: *NPF_EntryAdd()*; *NPF_A_Function()*

4.6 Enumerated Values

Enumeration values follow the convention defined in section 4.4 with the addendum that each enumeration value **MUST** have an explicit numerical value associated with it and enumeration value names **MUST** contain only capital letters. Enumerations **SHOULD** be used instead of *#defines* to achieve strong typing.

```
typedef enum{NPF_SATURDAY = 1, NPF_SUNDAY = 2} NPF_bestDays_t;
```

4.7 Abbreviations

Variable names should fully describe the entity the variable represents. When naming variables, state in words what the variable represents and try to create a concise but not cryptic abbreviation. A good mnemonic name generally speaks to the problem rather than the solution. A good name tends to express the ‘what’ more than the ‘how’.

Industry-standard abbreviations **SHOULD** be used where possible and the following guidelines followed:

- Don’t abbreviate by removing just one letter from a word, use July not Jul
- Always use the same abbreviation for the same concept, if using Num do not use No.
- Make them pronounceable.
- Use a thesaurus to help resolve conflicts.

Alloc	Allocate
Avg	Average
CB	Callback
Max	Maximum
Min	Minimum
Mux	Multiplexed
Rx	Receive
Sync	Synchronization
Tx	Transmit
Xc	Cross-connect.

Table 1 Abbreviation Examples

4.8 Function, Type, and Variable Name Lengths

All NPF defined functions, types and variables shall observe the ANSI C89 standard [1] for uniquely discriminating function names by being unique in the first 31 characters of the name.

4.9 NPF Function Name Composition

Function names **SHALL** be composed of attributes in big-to-small order, ending in a verb that characterizes its operation, for example

```
NPF_<Module><Feature><Verb> = NPF_Ipv4UnicastAlarmGet()
```

Components and features **MAY** be followed by subcomponents and sub-features whenever applicable. Consecutive duplicate attributes **SHOULD** be combined (e.g., NPF_Ipv4UnicastInit is the device initialization function, not NPF_Ipv4UnicastInitInit).

4.9.1 NPF Module

```
NPF_<Module><Feature><Verb> = NPF_Ipv4UnicastAlarmGet()
```

The names of functions, type definitions, and constants affiliated with a component type **SHOULD** specify the component name immediately after the NPF_ prefix. For example,

```
NPF_Ipv4UnicastAlarmGet().
```

4.9.2 NPF Features

`NPF_<Module><Feature><Verb> = NPF_DeviceAlarmGet()`

Feature provides the identifier for what is to be controlled by this function. Many of these names will be component specific. APIs SHOULD use the following list of terms where possible for describing features.

Prov	Provision or configure the component
Status	Status of the component
Alarm	Alarm
AlarmPersist	Alarm persistency
Alloc	To allocate memory for data structures
Children	Identify children
Parents	Identify parents
Version	To get the API version of a component

Table 2 Feature examples

4.9.3 NPF Verbs

`NPF_<Module><Feature><Verb> = NPF_DeviceAlarmGet()`

The function verb identifies the action being performed on the feature of a component.

Function verbs often come in pairs such as add/remove and begin/end. Verbs MAY also be concatenated, for example to get all the enabled elements of a module `NPF_ModuleFeatureEnabledGet()`

Clear	Reset latched status indications
Conv	Convert (e.g., to/from interrupt id)
Get	Get information
Set	Set information
Start	Start a one-shot or periodic operation
Next	Get the next item
Create	Initialize/Create
Destroy	Finalize/Destroy
Reinit	Reinitialize/Recreate
Alloc	Allocate resources
Free	Free resources
Enable	Enable a component
Disable	Disable a component

Table 3 Verb examples

5 Data Types

In this section, common data types will be defined, as well as guidelines to follow in definition of more complex types.

5.1 Basic Data Types

This section lists the of basic data types that **MUST** be used in NPF APIs. When APIs are approved the associated data types **MUST** also be approved.

Rules of data types:

- Data structures packing and endianness is generally considered beyond the scope of this document as source code compatibility can be achieved without awareness of either one. Certain types (e.g. for IP Addresses) **MAY** have a specified endianness but most types shall not.

Basic Data Types	Notes
NPF_boolean_t	An enumeration of NPF_TRUE (value 1) and NPF_FALSE. (value 0)
NPF_char8_t	An eight bit wide character format.
NPF_uchar8_t	An eight bit wide unsigned character format suitable for passing arrays of bytes.
NPF_int16_t	A sixteen bit signed integer format.
NPF_int32_t	A thirty-two bit signed integer format.
NPF_int64_t	A sixty-four bit signed integer format.
NPF_uint16_t	A sixteen bit unsigned integer format.
NPF_uint32_t	A thirty-two bit unsigned integer format.
NPF_uint64_t	A sixty-four bit unsigned integer format.
NPF_float32_t	A thirty-two bit signed floating point format.
NPF_float64_t	A sixty-four bit signed floating point format.

Table 4 Basic NPF Data Types

5.2 Common Data Types

This section lists a set of basic data types that are common throughout the NPF APIs. These data types are built upon the basic data types defined above.

Common Data Types	Notes
NPF_error_t	A 32-bit wide numerical error code value.
NPF_callbackHandle_t	A handle used to identify a callback function.
NPF_callbackType_t	A per-API family enumeration used to discriminate callback types.
NPF_correlator_t	An opaque 32-bit wide value that API users use to contain opaque per-function invocation data that is returned during callbacks.
NPF_userContext_t	An opaque 32-bit wide value that API users use to contain opaque per-callback function invocation data that is returned during callbacks.
NPF_eventMask_t	A per-API family bit-mask, used to indicate which events an application wishes to receive.
NPF_errorReporting_t	An enumeration used to indicate the degree of error reporting desired from callback functions when registering for them.

Table 5 Derived NPF Data Types

5.3 Rules for Construction of New Data Types

Construction of new data types MUST follow the following guidelines:

- All data types defined by official NPF specifications MUST use the `NPF_` prefix as described in section 4.1.
- All NPF constructed data types MUST be constructed out of types defined in sections 5.1, 5.2, or structures built using these basic types.
- Data types that will be passed by value should be of reasonable size. Reasonable size is a necessarily vague term as it takes into account multiple factors, including but not limited to the frequency a type will be used, the size of the members of the type (if a struct), and the expected environment (e.g. a FAPI level call vs. a higher level API call).
- When passing larger amounts of data, or passing a sizable amount of data very frequently, it is recommended that a pointer be passed rather than passing the data itself on the stack.

6 Parameter Passing

This section describes conventions used in passing parameters to API functions. This includes classification of parameters types for ordering in the function definitions, options for when to pass parameters by value or by reference, as well as standard behavior for treatment of in, out and in-out types of parameters. Memory allocation and release are described where applicable.

6.1 Scalar Arguments

Scalar arguments SHOULD be passed by value only; not by pointer.

6.2 Array Arguments

Array arguments SHOULD be accompanied by scalar argument(s) that indicate the dimension(s) of the array. The scalar argument(s) MAY be omitted when the array size is known and fixed.

6.3 Resource Handles

Many API functions will use “handles” to reference structures resident in the memory of the caller or the callee, but not both. Use of handles in NPF Software API specifications SHOULD adhere to the following guidelines:

- The API definition should assume that only the entity that generated the handle knows what the actual content of the handle value is.
- The API definition should put no assumption on the handle value. Especially the handle value should not be assumed that they are globally unique. (Handle is only required to be unique within the entity and the context in which the handle is generated.)
- The API should be designed so that any necessary resources associated with the handle be allocated by the entity that generated the handle.
- The API should be designed so that the removal of handle be done by the entity that has generated the handle, which enables the entity to de-allocate any resources related to the handle.

6.4 Memory Ownership

Memories that are used to hold values that are passed around as parameters are initially “owned” by the side (caller or callee of a function) that allocated them. An owner of a memory is responsible for de-allocating the memory when the memory is no longer in use.

In general, passing parameters through NPF defined functions will not change the ownership of the memory that is passed as parameters. For example, if a caller of an NPF defined function allocated a block of memory and passed a pointer to the memory as a parameter to the function, the memory ownership and the responsibility to de-allocate the memory remain with the caller.

This is the default behavior, and any NPF defined functions that involve changes to memory ownership MUST clearly state it in the function definition.

6.5 NPF_IN/NPF_OUT/NPF_IN_OUT Parameters

In order to better document the function of each parameter in NPF defined API function calls, a set of null macros named NPF_IN, NPF_OUT, and NPF_IN_OUT SHALL be used with all NPF API function calls. These macros MUST be included in front of each parameter of an NPF specified function so as to guide

the API client in usage. These macros are associated with a specific memory management and ownership scheme for parameters. The following subsections describe these schemes.

Note: In the following table, rows with “Design” keyword describes the API design guideline, and rows with “Impl” keyword describes implementation guideline for such API.

6.5.1 NPF_IN Scheme

Parameter Type	Guideline Type	Guideline
Scalar	Design	Scalar <code>NPF_IN</code> parameter MUST be passed by value.
	Impl.	The value MUST be assigned by the caller.
Compound	Design	Compound type <code>NPF_IN</code> parameter can be passed either by value or by pointer. Guideline for API designers on whether value passing or pointer passing SHOULD be used, is set by the “Parameter Passing” section of the SwAPI Software Convention document. When passed by pointer, <code>const</code> modifier SHOULD be used to protect the value of the parameter from being modified by the callee.
	Impl.	The caller MUST assign the value of the parameter. When pointer passing is used, the passed pointer MUST be pointing to a valid buffer at the point of function call. Callee MUST not change the value of the parameter (the buffer), or the implementation will be considered NPF API non-compliant.

6.5.2 NPF_OUT Scheme

Parameter Type	Guideline Type	Guideline	
Scalar	Design	Scalar <code>NPF_OUT</code> parameter MUST be passed by a pointer.	
	Impl.	The pointer MUST be pointing to a valid buffer at the point of method invocation. The content of the buffer, if any, pointed by the pointer, will be ignored by the callee. (The buffer does not need to be initialized.) The callee will set the value to the buffer, which the pointer is pointing to.	
Compound	Pattern 1	Design	Compound type <code>NPF_OUT</code> parameter MUST be passed by pointer.
		Impl	The pointer MUST be pointing to a valid buffer at the point of method invocation. The content of the buffer, if any, pointed by the pointer, will be ignored by the callee (The buffer does not need to be initialized.) The callee will set the value to the buffer, which the pointer is pointing to.
	Pattern 2	Design	Compound type <code>NPF_OUT</code> parameter is passed as a pointer(2) to a pointer(1).
		Impl.	The pointer(2) is pointing to a valid buffer for pointer(1), at the point of method invocation, The pointer (1) does not need to be pointing anywhere, and its value will be ignored by the callee. The callee will set the pointer(1) to point to a valid buffer that contains the value of the <code>NPF_OUT</code> parameter. The caller does not change the value of the <code>NPF_OUT</code> parameter (the buffer that the pointer(1) is pointing to).

Pattern 2 for `NPF_OUT` parameters **MUST NOT** be used.

6.5.3 NPF_IN_OUT Scheme

Parameter Type	Guideline Type	Guideline	
Scalar	Design	Scalar <code>NPF_IN_OUT</code> parameter MUST be passed by a pointer.	
	Impl.	The pointer MUST point to a valid buffer when the function is called. The caller MUST set a valid input value to the buffer, which will then be overridden by the callee with an output value	
Compound	Pattern 1	Design	Compound type <code>NPF_IN_OUT</code> parameter MUST be passed by pointer.
		Impl	The pointer MUST be pointing to a valid buffer at the point of method invocation. The caller MUST set a valid input value to the buffer, which will then be overridden by the callee with an output value.
	Pattern 2	Design	Compound type <code>NPF_IN_OUT</code> parameter is passed as a pointer(2) to a pointer(1).
		Impl.	The pointer(2) is pointing to a valid buffer for pointer(1), at the point of method invocation, The pointer (1) is pointing to a valid buffer, which contains a valid value as an input parameter, at the point of method invocation. The callee can either do the following <ul style="list-style-type: none"> - Set the pointer(1) to point to a valid buffer that contains the output value of the parameter. In which case the caller does not modify the value of the buffer that the pointer(1) is pointing to. - Set the output value to the buffer that the pointer(1) is pointing to. In which case, the callee will overwrite the values set by the caller. The caller is free to modify the buffer that pointer(1) is pointing to, after the method is completed.

Pattern 2 for `NPF_IN_OUT` parameters **MUST NOT** be used.

6.6 Coherent State Image of Dynamic Elements

Some functions will be designed to return a data set whose size is not known to the caller. If the set represents the state of a dynamic entity, i.e. one that can change state at any time, it **SHOULD** be returned in a way that guarantees that the caller receives a complete and self-consistent image of the data as it was at some instant in time while the call was being processed.

6.7 Support for Local Parameters/Avoidance of Complex Locking

Applications **MUST** be allowed to pass parameters that are contained in local variables. That is, an application should be able to pass in a pointer to a local variable in cases where it needs to provide a string or other array via an NPF API. Similarly, it is important to not require resources to be locked across several invocations of functions. Thus, for all normal parameter passing, it is incumbent on the callee to make a copy of any array or pointer handles before returning from a function call. This is the default behavior; any exceptions **MUST** be explicitly documented. In general, APIs must protect themselves against dangling handles and lingering memory allocations for structures that are no longer in use.

It is noted that there may be APIs where the amount of data expected to be transferred across it is high enough that a forced copy will cause a performance hit. In such scenarios it is acceptable to use a buffer handling semantic that reduces copies at the cost of delaying release of a buffer beyond the duration of a function call. This is further addressed in the Packet Handler API.

6.8 API Signature Guidelines

Strong typing SHOULD be used in defining API function calls. The following subsections set forth guidelines for defining different kinds of API function calls.

6.8.1 Multi-Field Inputs

For API function calls that are reasonably expected to accept a related set of information all at the same time, a struct SHOULD be used to carry that information, particularly in cases where multiple instances of the information are expected. Figure 7 shows an example of what such an API function call might look like.

```
NPF_error_t NPF_EntryAdd( NPF_IN NPF_tableHandle_t    myTableHandle,
                          NPF_IN NPF_entry_t         routes[],
                          NPF_IN NPF_ushort16_t      entryCount);
```

Figure 1 Example strongly typed API function call accepting an array of related items

6.8.2 Multi-Field Queries

When querying for information such as a set of related counters for an interface, the function doing so SHOULD use a well defined structure for the information, accepting a handle indicating the device being queried and providing an out parameter with a pointer to a structure that contains the counters in question, with a return parameter indicating whether the handle type matches the type of the call. This optimizes for having a simple, well-defined approach for querying multiple counters while minimizing the number of function calls required. It is noted that this approach can result in more counters being retrieved than is strictly needed, however, this is considered acceptable given that the overall cost of such a function call is very low compared to issues such as overhead for accessing an NPE. For those queries where a very large collection of related items may be retrieved, API writers SHOULD partition the items into subsets, and then define functions for retrieving each of the subsets.

```
NPF_error_t NPF_StatsQuery(NPF_IN  NPF_device_t      myDeviceHandle,
                           NPF_OUT NPF_counters_t    *counters);
```

Figure 2 Example strongly typed API function call returning a set of related items

6.8.3 Control Interfaces

Functions used to deliver configuration information for interfaces and other devices are expected to typically require the ability to manipulate a single setting without having to reset the entire configuration of a device. As such, control interfaces SHOULD typically be written to manipulate a single characteristic of a controlled device. In order to facilitate re-use and avoid function proliferation, functions SHOULD be made generic for individual functions across a range of controlled interface types where possible. Thus, figure 3 is an example of how the Foo attribute could be set on a wide variety of interface types without requiring multiple `SetFooFor{FooInterface|BarInterface|BazInterface}()` functions.

```
NPF_error_t NPF_FooSet( NPF_IN NPF_device_t      myDeviceHandle,
                       NPF_IN NPF_foo_t         fooValue);
```

Figure 3 Generic Control Interface Example

6.9 Packet Buffer Handling

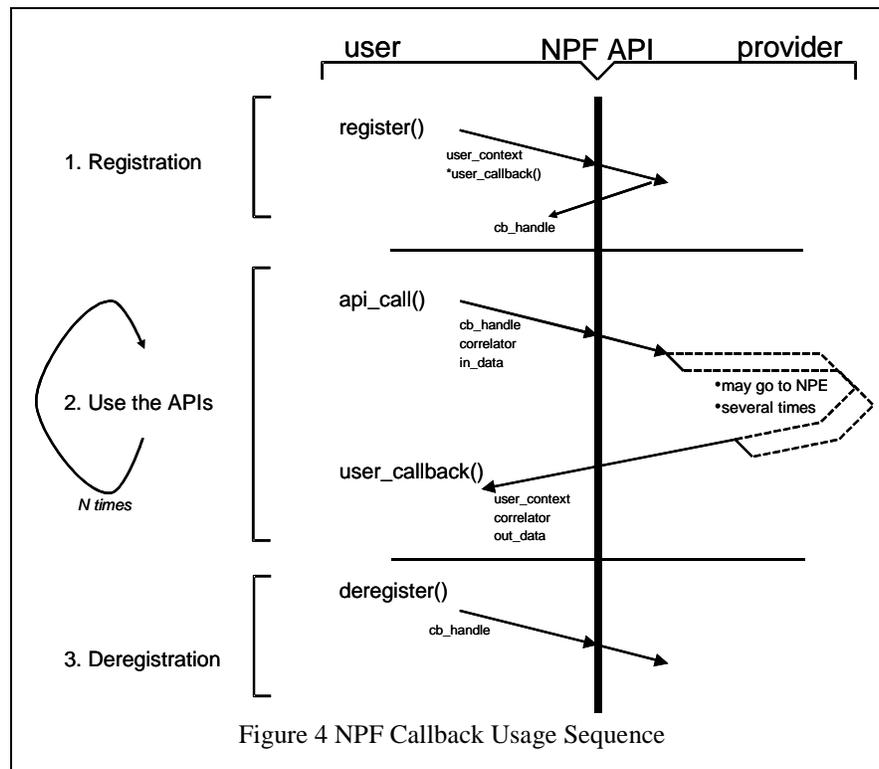
Guidelines and semantics for packet buffer ownership and handling are described in the Packet Handler API document.

7 Function Invocation Model, Events and Completion Callbacks

7.1 API Completion Callbacks

The function invocation model for NPF APIs is based on asynchronous callbacks, where there is a single callback for each function call. The completion of work associated with an API function call is not indicated by the return of the call, but by the invocation of a separate completion callback function by the callee to the caller. This allows more flexibility in that the caller is not blocked waiting for the result. This enables more parallelism to be achieved, while still allowing synchronous behavior to be easily layered on top of the asynchronous callbacks if desired.

Note that in some cases, the work can be completed before the original call returns. In those cases, the completion callback may be called before the original call returns, whether on the same thread or a different worker thread, which is implementation dependent. In a typical multi-threaded environment, the work performed on behalf of an asynchronous request may be accomplished by a separate worker thread (or threads). This thread, unless synchronized with the original call, may complete its work before or after the original call completes. Rather than constrain the implementation to synchronize or perform its work on the original call thread, no guarantee is made for when the completion callbacks can be called. In addition, because an application must design for true asynchronous completion callbacks due to this lack of guarantee, it is not useful to allow some implementations to optionally return and indicate that a completion callback is unnecessary. However, an implementation is free to perform completion callbacks on the original call threads if the work can be performed immediately and to reduce context switches.



There need not be a 1:1 relationship between an asynchronous request call and a completion callback. A single request call may generate multiple asynchronous response callbacks. Partial completions may happen, for example, when there are multiple input arguments in the request, and the implementation

chooses to process and complete the arguments independently. The application SHOULD keep track of outstanding and partially completed requests, to ensure detection of full completion.

Some error conditions may result in a return without invocation of the completion callback. This is further described in the Error Handling section.

Completion callbacks for API functions are grouped into separate categories. The application will register a completion callback function for each of the categories of completion callbacks that are of interest. A completion callback handle is provided to the application upon successful registration. The application provides this completion callback handle when using API calls for that type. The application may register and de-register completion callbacks at initialization and shutdown respectively, or as needed during the execution of the application.

The application provides two types of application context information. A `NPF_userContext_t` is provided at callback registration time. In addition, a `NPF_correlator_t` is provided at API function call time. When the completion callback is called for a particular call, the application will receive the callback type, and both the `NPF_userContext_t` and the `NPF_correlator_t` contexts, as well as result information. This information allows the application to associate a completion callback call with a specific instance of the original function call. The callback type MAY indicate a specific type within a category.

The application can also provide an error reporting level with the API function call to indicate interest in receiving the completion callback (e.g. call back only if an error occurs, always call back, never call back). While an API implementation SHOULD provide best effort to deliver callbacks, delivery of callbacks is not guaranteed. Thus it is the application's responsibility to protect itself against duplicated or lost callbacks.

Figure 4 shows the basic model and usage of NPF asynchronous APIs as defined in this document.

Below are examples of the registration, de-registration and completion callback functions, where `Xxxx` will represent a category for the callback and `callbackType` will represent a specific type in a category. `Xxxx` and `callbackType` will be replaced with category and type definitions as defined by each API. Also included is a convention for API function parameters that are required for callback support. Note: the registration, de-registration and completion callback functions are synchronous.

7.1.1 NPF_XxxxRegister

This function allows the application to register a completion callback function for the related callback category, and to associate a unique callback handle as well as application context.

Signature: `NPF_error_t NPF_XxxxRegister(NPF_IN NPF_userContext_t, NPF_IN NPF_XxxxCallbackFunc_t, NPF_OUT NPF_callbackHandle_t *)`

Parameters In: `NPF_userContext_t`
`NPF_xxxxCallbackFunc_t`

Parameters Out: `NPF_callbackHandle_t`

Return Values: `NPF_NO_ERROR`
`NPF_E_BAD_CALLBACK_FUNCTION`
`NPF_E_CALLBACK_ALREADY_REGISTERED`
`NPF_E_UNKNOWN`

7.1.2 NPF_XxxxDeregister

This function allows the application to de-register the callback function that is associated with this callback handle.

Signature: NPF_error_t
 NPF_XxxxDeregister(NPF_IN NPF_callbackHandle_t)

Parameters In: NPF_callbackHandle_t

Parameters Out: None

Return Values: NPF_NO_ERROR
 NPF_E_BAD_CALLBACK_HANDLE
 NPF_E_UNKNOWN

The callback routine might be called after the deregistration function has been invoked, but the API implementation SHALL guarantee that the callback function is not called after the deregister function has returned.

7.1.3 NPF_XxxxCallback_FUNC

This is the function template for function completion callbacks. These callbacks are asynchronously invoked in response to NPF API calls.

The context and correlator parameters come from the callback function registration function and the original function call, respectively. The callback data is a structure that contains result information about the original function call.

```
typedef void (*NPF_xxxxCallBackFunc_t)( NPF_IN NPF_userContext_t,
                                         NPF_IN NPF_correlator_t,
                                         NPF_IN NPF_xxxxCallbackData_t)
```

7.1.4 Callback Data Structure

This section defines the data structure used to deliver result information in the context of a callback. The comment text below explains the parameters and structures and their usage.

```
typedef NPF_uint32_t NPF_xxxxErrorType_t; /* Error code (NPF_XXXX_E_YYYY) */
```

/*

- 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_xxxxCallbackData_t structure (below).

*/

```
typedef struct { /* Asynchronous Response Structure */
    NPF_xxxxErrorType_t error; /* Error code for this response */
    NPF_xxxxId_t resource; /* User's resource id for handle */
    union { /* Function-specific structures: */
        NPF_uint32_t unused; /* Default */
        NPF_xxxxHandle_t xxxxHandle; /* Handle from NPF_XxxxTableCreate*/
    } u;
} NPF_xxxxAsyncResponse_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_XxxxAsyncResponse_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_xxxxCallbackType_t  type;           /* Identifies the function called */
    NPF_boolean_t          allOK;         /* TRUE if all requests completed OK*/
    NPF_uint32_t           numCallbackResp; /* Number of responses in array */
    NPF_xxxxAsyncResponse_t *resp;       /* Array of response structures */
} NPF_xxxxCallbackData_t;

```

Some NPF APIs may support a form of batched request that lets an application pass a variable-length array of “request elements” to the API implementation. Examples: the Ipv4 route add function can take an array of prefix/length parameters; the Interface Create API takes an array of interface specifications, and creates many with a single call.

The callback mechanisms should be defined similarly for these APIs. Points they will have in common:

1. The implementation generates one “response element” for each “request element”. A response element contains a status code and something that identifies the request element to which it belongs. Examples of identifiers: prefix/length values (for route add and route delete); next hop index (for next hop entry add and delete); IP address and interface (for resolution table entry add and delete).
2. The implementation MAY pass an array of response elements to the callback function at any time after the API function is called. For any API function call, the total number of response elements passed to the callback function should be exactly the same as the number of request elements passed to the API function. There should be exactly one status code returned for each request element.
3. How many response elements are passed each time the implementation invokes the callback function? How many invocations of the callback function is the implementation

allowed to make in order to return all the response elements for a single API function call? These are for the implementor to decide. The minimum number of callback invocations is one (unless the `errorReporting` parameter causes responses to be suppressed); the maximum is the number of request elements passed in the API function call.

4. An NP Forum API module, such as the Ipv4 Unicast Forwarding API or the Interface Management API, MAY support the “AllOK” option, defined as follows:
 - a. The structure defined by the API to be passed to the callback function contains a variable called “AllOK”, with values of TRUE and FALSE.
 - B. IF the API implementation passes the complete list of response elements in a single invocation of the callback function, and IF the status code for every response element is “no error”, the implementation SHALL return `AllOK = TRUE` and omit the array of response elements (i.e. pass back an array length of zero).
 - C. If, at the time of callback invocation, a response element for any of the request elements indicates an error or is missing, the implementation SHALL return `AllOK = FALSE` and pass an array of response elements that includes responses for all request elements for which the completion status is known.

7.1.4.1 Callback Type and Error Fields

These members of the callback data structure defined above are associated with the original function call that caused the callback to be made. They are used to determine which member type of the union is present.

Within each API family a specific enumeration SHALL be defined for the callback type and error type fields that are described above. The callback type field shall correspond on a 1:1 basis with the original function calls that invoke callbacks. The error field values may be standard across a set of functions within an API family or specific to a particular function.

7.1.4.2 Guidelines for definition of callback union structures

Callback union structures defined for specific API function calls may need to include additional data as part of the structures included in the union. Since the data will be present as part of a union, large data may seriously affect memory requirements for these structures as unions allocate the amount of memory needed for the largest member of the union irrespective of the actual member type in use. In such cases, a pointer to the data may be a better choice to define as a member of one of the structures that make up the union.

Vendors can define their own callback structure types by adding proprietary fields within the union to provide additional data. These are not be defined by NPF.

7.1.5 API Function Signature Requirements

API function calls use three parameters to support asynchronous completion callbacks. Each API function will correspond to a defined callback category. `NPF_callbackHandle_t` is provided to the application upon registration. `NPF_correlator_t` is an application context. `NPF_errorReporting_t` indicates whether the application wishes to receive a completion callback or not, or only upon errors. `NPF_errorReporting_t` is an enumeration containing `NPF_REPORT_ALL`, `NPF_REPORT_NONE`, and `NPF_REPORT_ERRORS`. If the function is creating a

resource, then “other function parameters” MUST include as input a resource id parameter, declared as type `NPF_xxxxId_t`, for identifying lost or duplicate callbacks.

Signature: `NPF_error_t NPF_Xxxx<api function name> (`
`NPF_IN NPF_callbackHandle_t,`
`NPF_IN NPF_correlator_t,`
`NPF_IN NPF_errorReporting_t,`
`<...other function parameters...>)`

Required Parameters In: `NPF_callbackHandle_t`
`NPF_correlator_t`
`NPF_errorReporting_t`

Required Parameters Out: `None`

7.1.5.1 Error Reporting and Callbacks

The `NPF_errorReporting_t` enumeration defines three values: `NPF_REPORT_ALL`, `NPF_REPORT_NONE`, and `NPF_REPORT_ERRORS`. When invoking a function with an asynchronous callback in the NPF APIs one of these values MUST be passed in. These values cause the following behavior on the part of a compliant implementation:

- `NPF_REPORT_ALL` will cause all function calls associated with an asynchronous callback to result in a callback, whether the function succeeded or not. The only exception to this is function calls that immediately return an error code instead of `NPF_NO_ERROR`.
- `NPF_REPORT_NONE` causes function calls associated with asynchronous callbacks to never result in a callback. This value is useful when the results of a function call do not matter.
- `NPF_REPORT_ERRORS` causes function calls associated with asynchronous callbacks to only callback to an application when an error occurs as part of the execution of the call. Note that function calls that immediately return an error code will not later result in a callback.

Certain types of function calls (e.g. statistics queries) are nonsensical when used with `NPF_REPORT_NONE` or `NPF_REPORT_ERRORS`. Such functions MUST immediately return an error code when invoked with these values and MUST be clearly documented as such.

7.1.6 Reentrancy

The NPF API function calls consist of two categories of invocations:

- 1) Synchronous API function calls (e.g. various register, deregister API function calls) and asynchronous API function calls (e.g. API function calls for manipulating the various tables on the forwarding plane).
- 2) Completion callbacks and event notifications

API function call reentrancy means that an application can invoke a function call a second time, before its first invocation has completed. Similarly, callbacks or event notification functions reentrancy means that a second callback or event notification invocation is handled, before its first invocation has completed

Synchronous function calls SHOULD be reentrant. Using reentrancy with synchronous function calls is rather rare. A synchronous function call can be invoked before its previous invocation completed only by application code running at a higher priority level than the first invoking code.

Reentrancy is used frequently with asynchronous function calls. For example, an application can invoke a function call to add routes a second time before its first invocation has completed. In order to ensure applications portability to any NPF API implementation, asynchronous function calls MUST be reentrant.

7.2 Lost or Duplicate Callbacks

Because an API implementation provides only best effort delivery of callback responses, delivery of callbacks is not actually guaranteed. It is the application's responsibility to handle lost or duplicate callbacks.

In order to allow an application to recover in these cases, an API function call which requests the creation of one or more new resources **MUST** include a user-generated resource id parameter for each new resource. This resource id is a non-zero integer that is unique for a given resource type, for the life of the resource.

The implementation can detect duplicate creation requests by checking for a previously existing resource id. A user may then safely repeat the request to create a resource when they suspect the original create callback was lost. Duplicate create callbacks can be ignored. Similarly, missing or duplicate callbacks for API's which *destroy* a resource may be handled by simply repeating the request or ignoring the duplicate.

Because handling lost or duplicate callbacks requires adding a new resource id argument to certain handle creating calls, it is a *backwards incompatible* change. The "NPF API" and Task Group major revision numbers must therefore be incremented to Version 2.X.

7.2.1 Resource Id Types

The data type for the resource id is defined per API and **MUST** be of the form NPF_XXXXId_t. An API may define more than one resource id type, as desired. However, there should be at most one resource id type for each handle type. A single resource id type may be used by different handle types, but not the other way around. A generic example declaration would be:

```
typedef NPF_uint32_t  NPF_XXXXId_t;
```

7.2.2 Create and Destroy Error Codes

When a duplicate create request is detected, the NPF_E_RESOURCE_EXISTS error is returned and no new resource is created. Duplicate request errors may be reported by the implementation at either API function call invocation time or completion callback time. All callbacks for the resource type **MUST** include the resource id.

Similarly, when a duplicate request to destroy or free a resource is detected, the NPF_E_RESOURCE_NONEXISTENT error is returned to the called. Duplicate destroy errors also may be reported at either API function call invocation time or completion callback time.

7.2.3 Querying Resource Id's

The NPF Implementation Agreement **MAY** provide a query function for a resource so that a user may determine whether a given resource id already exists. If it does not already exist, the create request may be repeated. The query function returns an array of <handle,resource id> pairs for all resources of a given type. Query functions are asynchronous, to allow a user program to make progress with partial completion of the query.

7.2.4 Atomic Modification of Resource State

In order to allow clean recovery from missing or duplicate callbacks, all API's which modify the state of a resource **MUST** be atomic. Partial or incremental modification of resource state is not permitted by any API.

7.3 Event notification

Any applications interested in events occurring on the network processor may register for notification of these events. The events may or may not be related to invocation of NPF API calls, and may include indications of such occurrences as a link going down, or an IP address changing. Events are organized in separate categories and an application can register for individual events, or for an entire category of interest.

A handle is provided to the application upon successful registration. The application provides this handle when unregistering for the events. The application may register and de-register for events at initialization and shutdown respectively, or as needed during the execution of the application.

An application can register a handler to receive selected events by setting a bit in the `NPF_eventMask_t` parameter for each event it wishes to receive, when it calls the event registration function. A mask with all bits set (`0xFFFFFFFF`) selects all events of the particular category. If the application wishes to change the selection of events for a particular handler function, it may call the event registration function again with the same handler function address and context, but with a different event selection mask. The events enabled are those whose bits were set in the most recent registration function call for a particular handler function/context pair. Re-registration of one function/context pair does not change the event selection for any other function/context pair that might also be registered at the time.

The application provides context information, of type `NPF_userContext_t`, at event registration time. The `NPF_correlator_t` used with API function calls is unnecessary in the event notification model, as no invocations are made that can be correlated. When the event notification is made for a particular event, the application will receive the event information, and the `NPF_userContext_t` context information.

Below are templates of registration, de-registration, and event call functions, where `Xxxx` will represent a category or a specific event in a category. `Xxxx` will be replaced with category and event definitions and structures as defined by each API. Note: the registration, de-registration and event call functions are synchronous.

7.3.1 NPF_XxxxEventRegister

This function allows the application to register a function for the event or event category, and associate a handle with the registration. Registration accomplishes two things: It registers the event notification function (the “handler”) to the API, and also enables notifications for the events selected by the bits that are set in the `NPF_eventMask_t` parameter. Note that the implementation may begin to invoke the event handler before returning from the registration function.

Signature: `NPF_error_t NPF_XxxxEventRegister(`
`NPF_IN NPF_userContext_t,`
`NPF_IN NPF_xxxxEventCallFunc_t,`
`NPF_IN NPF_eventMask_t,`
`NPF_OUT NPF_callHandle_t *)`

Parameters In: NPF_userContext_t
 NPF_xxxxEventCallFunc_t
 NPF_eventMask_t

Parameters Out: NPF_callHandle_t *

Return Values: NPF_NO_ERROR
 NPF_E_BAD_CALLBACK_FUNCTION
 NPF_E_CALLBACK_ALREADY_REGISTERED
 NPF_E_UNKNOWN

7.3.2 NPF_XxxxEventDeregister

This function allows the application to de-register the event notification function that is associated with this callback handle.

Signature: NPF_error_t NPF_EventDeregister(NPF_IN NPF_callHandle_t)

Parameters In: NPF_callHandle_t

Parameters Out: None

Return Values: NPF_NO_ERROR
 NPF_E_BAD_CALLBACK_HANDLE
 NPF_E_UNKNOWN

The event routine might be called after the deregistration function has been invoked, but the API implementation SHALL guarantee that the event function is not called after the deregister function has returned.

7.3.3 NPF_xxxxEventCallFunc_t

This is the event notification function format. Xxxx indicates either a category of events, or a particular event. This function is invoked when the related event happens. NPF_userContext_t is the original value provided by the application during event registration. NPF_xxxxEventData_t contains an event type indicator and a union of event types provided by a particular API specification.

NPF_xxxxEventArray_t is a structure that contains an array of NPF_xxxxEventData_t structures, accompanied by a scalar describing the array length.

Definition:

```
typedef void (*NPF_xxxxEventCallFunc_t)( NPF_IN NPF_userContext_t,
                                         NPF_IN NPF_xxxxEventArray_t)
```

Definition:

```
typedef struct {
    NPF_uint16          numEventData; /* number of structures */
    NPF_xxxxEventData_t *eventData;
} NPF_xxxxEventArray_t;
```

Definition:

```
typedef struct NPF_xxxxEventData {
    NPF_xxxxEvent_t eventType;
    union {
        eventDataTypel_t c;
        eventDataTypel2_t d;
        <...>
    } u;
} NPF_xxxxEventData_t;
```

7.3.3.1 NPF_xxxxEvent_t

The type NPF_xxxxEvent_t is used to indicate the type of the structures returned in the union of event structures. The definition of this type is specific to each xxx API family and MUST be an enumeration of event types supported by that family.

Each API family SHALL define its own event type,. For instance, the Ipv4 Unicast Forwarding API defines:

```
typedef enum
  NPF_Ipv4Event { NPF_IPV4_ROUTE_TABLE_MISS = 0,
                 NPF_IPV4_NHR_UNREACHABLE = 1} NPF_Ipv4Event_t;
```

7.3.3.2 NPF_eventMask_t bit definitions

Each API SHALL define NPF_eventMask_t bits for all of its events, beginning with the rightmost bit of the mask. For example,

```
/*
  • Definitions for selectively enabling IPV4UC events
*/
#define NPF_IPV4UC_EV_PREFIX_TBL_MISS_ENABLE (1 << 0)
#define NPF_IPV4UC_EV_NEXT_HOP_TBL_MISS_ENABLE (1 << 1)
#define NPF_IPV4UC_EV_ADD_RES_TBL_MISS_ENABLE (1 << 2)
#define NPF_IPV4UC_EV_FIB_PREFIX_MISS_ENABLE (1 << 3)
#define NPF_IPV4UC_EV_LAST (1 << 3)
```

This document defines the following value for use by all API families:

```
#define NPF_EV_ALL_EVENTS_ENABLE (0xFFFFFFFF)
```

7.3.3.3 Guidelines for definition of event structures

Event structures defined for specific API function calls may need to include additional data as part of the structure. Since the data will be present as part of a union, large data may seriously affect memory requirements for these structures. In such cases, a pointer to the data may be a better choice to define as a member of the structure.

Vendors MAY define their own event structure types by adding proprietary fields within the union for additional data.

7.3.3.4 Example of event structures

Example of individual event structures:

```
typedef struct NPF_pscData {
    NPF_portID_t      portid;
    NPF_portStatus_t  status;
} NPF_pscData_t;
```

```
typedef struct NPF_pspData {
    NPF_portID_t      portid;
    NPF_uint64_t      speed;
} NPF_pspData_t;
```

Example of category event structures:

```
typedef struct NPF_itfData {
    NPF_itfEvent_t  eventType;
    union {
        NPF_pscData_t  pscData;
        <...>
        NPF_pspData_t  pspData;
    } u;
} NPF_itfData_t;
```

7.3.4 Event Notification Example

```

void NPF_InterfaceEventCallFunc(      NPF_IN NPF_userContext_t      context,
                                     NPF_IN NPF_itfDataList_t      data )
{
    switch (data.eventType) {
        case NPF_EVENT_PORT_STATUS_CHANGE:
            break;
        case NPF_EVENT_PORT_SPEED:
            break;
        default:
    }
}

int main() {
    NPF_callHandle_t callHandle;
    NPF_error_t retval;
    NPF_itfEventCallFunc_t eventCallFunc =
        & NPF_InterfaceEventCallFunc;
    retval = NPF_ITF_EventRegister((void*)getpid(),
                                   eventCallFunc, &callHandle);

    ...
    retval = NPF_ITF_EventDeregister(callHandle);
}

```

8 Error Handling

This section describes how error conditions are indicated to clients of the NPF API function calls.

8.1 Synchronous Error Returns

Error codes SHALL be returned synchronously from synchronous API function call invocations (such as callback registration) and from asynchronous API function call invocations where errors are detected before a completion callback is required. This may include error conditions such as invalid parameters or processing errors which occur in the context of the call or which prevent an asynchronous completion callback. Return values for API function call invocations shall be of type `NPF_error_t`.

8.2 Error Code Values

Valid error code values will be partitioned into ranges of values, with each API defining a corresponding range. See Appendix A. `NPF.h` for more information.

API Family	Error Code Range
Foundations (Common to all APIs)	0-99
Ipv4	100-199
Interfaces	200-299

Table 6 Assigned Error Code Ranges

8.2.1 NPF_NO_ERROR

This value MUST be returned when a function was successfully invoked. This value is also used in completion callbacks (see section 6) where it MUST be the only value used to signify success.

8.2.2 NPF_E_UNKNOWN

An unknown error occurred in the implementation such that there is no error code defined that is more appropriate or informative.

8.2.3 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.

8.2.4 NPF_E_BAD_CALLBACK_FUNCTION

A callback registration was invoked with a function pointer parameter that was invalid.

8.2.5 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.

8.2.6 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, as defined in this document).

8.2.7 NPF_E_RESOURCE_EXISTS

A duplicate request to create a resource was detected. No new resource was created.

8.2.8 NPF_E_RESOURCE_NONEXISTENT

A duplicate request to destroy or free a resource was detected. The resource was previously destroyed or never existed.

9 Compliance and Extensibility

In order to utilize the benefits of a common interface, a client application must be able to rely on the support of the interface by a conforming implementation. However, it is unreasonable to expect that an interface, once specified, will never change. Furthermore, some vendors may wish to provide their own useful extensions to the interfaces. The term “core” will be used to describe the common interfaces. The term “extended” will be used to describe additional or proprietary interfaces.

9.1 NPF-Defined Optional Functions and Data Structures

Certain NPF APIs MAY be defined to contain optional methods and/or data structures. Implementation of NPF APIs do not need to implement optional methods and data structures in order to be NPF compliant, but upon implementing all or a portion of the optional methods and data structures, all related optional methods and data structures required for proper functioning of the implementation must be implemented.

Example : If an optional method A assumes optional method B and optional data structure C for proper function, then if method A is implemented, method B and data structure C MUST be implemented.

The dependency among optional methods and data structures, if any, MUST be clearly documented in the API specification.

9.2 Revising NPF-Defined APIs

9.2.1 Version Number Assignment

NPF APIs can be seen as a collection of APIs where each API is defined by a TG. Since each APIs may evolve in difference paces, they each have their own versions. NPF APIs will also have a version number as a whole. (Mainly for vendors to express that they are NPF API 1.1 compliant rather than saying that they are Ipv4 1.2 Interfaces 1.3 Packet Handler 1.0 Diffserv 1.0 compliant.) The versioning of APIs MUST adhere to the following rules:

- Major version number will be shared among all NPF APIs. Major version numbering will be controlled by the Foundations TG with consensus from the entire WG.
- Minor version number for each API will be assigned by each TG defining the API.
- Minor version number for NPF APIs as a whole will be assigned by the Foundations TG.
- When major version number changes, all NPF APIs’ major version number will change accordingly (even if there is no change to the spec.).

A version of NPF APIs as a whole will be defined as a collection of versions of each separate APIs. Such version will be defined by the Foundations TG with consensus from the entire WG.

Example : NPF API version 1.1 can be defined as:

```

Foundations version 1.0
Ipv4 version 1.2
Interfaces version 1.3
Packet Handler version 1.1
FAPI 1.0
Diffserv 1.0

```

9.2.2 Version Number Checking

An implementation of an NPF API must define a macro with the following signature to express the version of API that it supports:

```
#define NPF_<MODULE>_<MAJOR_VERSION>_<MINOR_VERSION>_COMPLIANT
```

The macro can be used to check the version of API that an implementation supports at compilation time. It also can be used to do selective compilation.

An implementation can support multiple version of the same API if there is no conflict between them.

Example :

```
#define NPF_IPV4_1_2_COMPLIANT
#define NPF_IPV4_1_3_COMPLIANT
#define NPF_IPV4_1_4_COMPLIANT
```

means that the implementation supports Ipv4 1.2, 1.3 and 1.4 spec. It also suggests that there has been an alteration between 1.1 and 1.2 that broke backward compatibility.

Note : See 4.9.1 for definition of <MODULE>

9.2.3 Revising Method and Data Structure

When revising an API, occasionally new parameters may need to be added to a function, data structure need to be changed, or new semantics may be applied to an existing function. In order to maximize backward compatibility and to avoid confusion, the following rules must be followed when revising an API.

- Addition of a completely new function or data structure is always allowed.
- Changing parameters of an existing function within a same major version is prohibited except for bug fixes. When parameters need to be changed, a different function name must be used.
- Changing semantic or behavior of an existing function within a same major version is prohibited except for bug fixes. When semantic or behavior need to be changed, a different function name must be used.
- Changing return value of an existing function within a same major version is prohibited except for bug fixes. When return value need to be changed, a different function name must be used.
- Addition of new return value to an existing function is allowed, but should be avoided if possible since it may break backward compatibility.
- Changing existing data structure within a same major version is prohibited except for adding members to a union, adding members to the tale of a structure, or for bug fixes. Changing data structure may break backward compatibility.
- Changing type definition within a same major version is prohibited except for bug fixes. When a new type is needed, a new type should be defined.
- Changing value assignment to a constant is allowed.
- Changing contents of an enumeration is allowed.
- All changes listed above that are permitted without incrementing the major revision number require incrementing the minor revision number.

There are no restrictions in terms of changes when a major version number changes.

9.3 Vendor proprietary extensions

Vendors that implement the NPF APIs are allowed to a certain extent to modify the NPF API definition and its behavior and still claim conformance. The type of modification that a vendor can perform over NPF APIs are strictly limited to the following.

- Addition of a completely new proprietary function or data structure is always allowed.
- Changing parameters of an existing function within a same major version is prohibited. When parameters need to be changed, a different function name must be used.
- Changing semantic or behavior of an existing function is prohibited. When semantic or behavior need to be changed, a different function name must be used.
- Changing return value of an existing function is prohibited except for bug fixes. When return value need to be changed, a different function name must be used.
- Addition of new return value to an existing function is allowed, but should be avoided if possible since it may break backward compatibility.
- Changing existing data structure is prohibited except for adding members to a union, adding members to the end of a structure, or for bug fixes. Changing data structure may break backward compatibility.
- Changing value assignment to a constant **SHOULD NOT** be done.

10 Multi-Vendor / Multi-Instance Support

In the implementation of the NPF APIs, it is possible that the same APIs needs to be integrated multiple times. For example, this may happen when integrating the same Functional APIs from different vendors or if a particular implementer decides to integrate multiple instances of the same API (e.g. a Packet Handler API instance per forwarding blade). These types of multi-vendor or multi-instance integration can lead to conflicts such as:

- Conflicts in structures, typedefs, enums, etc
- Function name conflicts

The resolution of this problem of multiple-defined symbols is left to the integrator of the NPF APIs. The Software Conventions document will NOT propose any normative solution considering that: (1) there are a number of possible solutions, and (2) most operating systems provide efficient ways to solve this problem.

11 Design and Implementation Guidelines

The NPF Software API working group is primarily concerned with specifying a set of interoperable APIs. As such, vendors have complete control over the internal design and implementation of software implementing these APIs. This section provides some informative guidelines for implementation. This section of this document is NOT considered normative.

11.1 Modularity

A module is a collection of data and the routines that act on the data. A module might also be a collection of routines that provides a cohesive set of services even if no common data is involved. A module in C is a source file. One goal of a module is to hide information. In general, it is suggested that modules implementing the NPF APIs reveal as little as possible about their inner workings.

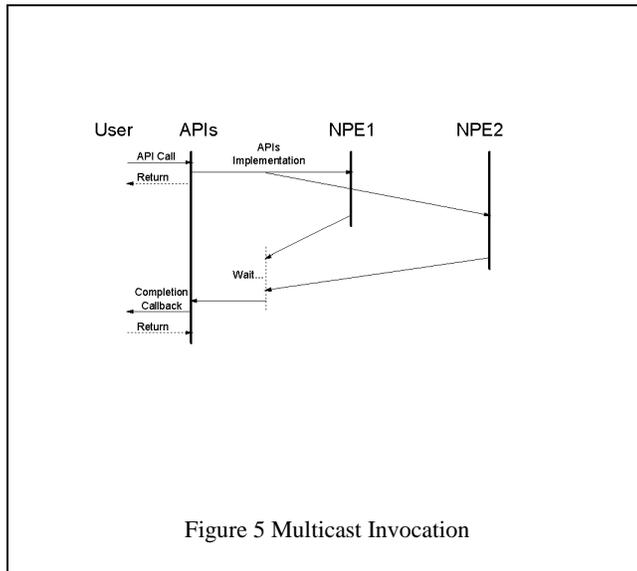
11.2 Multicast Invocations

In some architectures, a single services API function invocation can result in repeated calls to multiple forwarding elements to accomplish the result. For example, in a system with multiple network processors and a single FIB, a FIB update must be sent to all NPs. If not all NPs are successfully updated, does the request succeed or fail? How are the complex results of success and failure reported to the application?

The Software API Framework describes two API levels:

- Services API, whose purpose is to support applications while concealing details of the underlying system architecture;
- Functional API (FAPI), which addresses the specific functions of individual network processing elements.

Considering the example above: if the Services API must hide the fact of multiple NPs, an application call



to update a FIB must result in a single indication of success or failure, just as if there were only one underlying forwarding process to be notified of the update. The FAPI, likewise, addresses individual elements; so the FAPI-level request to update the Classification Element should also result in a single indication of success or failure.

This means that the coordination of multiple forwarding elements in a complex architecture is the function of software (“middleware”) residing below the Services API and above the FAPI. In no case shall any NP Forum API definition specify that requests are distributed to multiple elements with the possibility of multiple callbacks with differing results.

In the FIB example, middleware underlying the Services API must invoke the FAPI once to update each forwarding element. It SHOULD return success if the Services API request was valid and executable. If any of the individual FAPI calls fails, therefore, the failure should be a result of some incapacity of the system, not a problem with the nature of the request. The incapacity should result in an asynchronous notification to the application (through Services API Event Notification) of a failure of some part of the system, such as an interface going down. The middleware should increment error counters and generate event logs for diagnosis of the problem, as well as generate an event notification as described in this document.

11.3 Compatibility

Within a particular major version number, NPF APIs are only required to be compatible at the source code level and not at the binary code level.

12 References

- [1] “American National Standards Institute (ANSI), Standard for the C Language”, ANSI X3.159-1989.
- [2] Bradner, Scott, “Key Words for Use in RFCs to Indicate Requirement Levels”, IETF RFC 2119, Harvard University, March 1997.

Appendix A Header File: NPF.h

```

/* This header file defines typedefs, constants, and functions*/
/* that apply to all NPF Software Working group APIs.          */
#ifndef __NPF_H__
#define __NPF_H__

#ifdef __cplusplus
extern "C" {
#endif

#define NPF_IN
#define NPF_OUT
#define NPF_IN_OUT

/* This section defines base NPF types and will differ from */
/* platform to platform. The type shown here are based on */
/* Linux 6.2 on an x86.                                     */
typedef char                NPF_char8_t;
typedef unsigned char       NPF_uchar8_t;
typedef char                NPF_int8_t;
typedef short               NPF_int16_t;
typedef int                 NPF_int32_t;
typedef long long int       NPF_int64_t;
typedef unsigned char       NPF_uint8_t;
typedef unsigned short      NPF_uint16_t;
typedef unsigned int        NPF_uint32_t;
typedef unsigned long long int NPF_uint64_t;
typedef float               NPF_float32_t;
typedef long double         NPF_float64_t;

/* This section defines constructed NPF types and is          */
/* identical for all implementations of the NPF APIs.        */
typedef NPF_uint32_t        NPF_error_t;
typedef NPF_uint32_t        NPF_callbackHandle_t;
typedef NPF_uint32_t        NPF_correlator_t;
typedef NPF_uint32_t        NPF_userContext_t;
typedef enum
    NPF_boolean {NPF_FALSE = 0, NPF_TRUE = 1} NPF_boolean_t;
typedef NPF_uint32_t        NPF_eventMask_t;

/* Examples of other NPF types */
typedef NPF_uint32_t        NPF_Ipv4Address_t;
typedef NPF_uchar8_t        NPF_MAC_Address_t[6];

typedef enum
    NPF_errorReporting { NPF_REPORT_ALL      = 1,
                        NPF_REPORT_NONE     = 2,
                        NPF_REPORT_ERRORS   = 3 } NPF_errorReporting_t;

/*
    • NPF Event constants
*/
#define NPF_EV_ALL_EVENTS_ENABLE          0xFFFFFFFF

```

```
/*
 *   • NPF Error constants
 */
#define NPF_NO_ERROR (0)

#define NPF_FOUNDATIONS_BASE_ERR (1)
#define NPF_FOUNDATIONS_MAX_ERR (NPF_FOUNDATIONS_BASE_ERR + 98)

#define NPF_E_UNKNOWN (NPF_FOUNDATIONS_BASE_ERR)
#define NPF_E_BAD_CALLBACK_HANDLE (NPF_FOUNDATIONS_BASE_ERR + 1)
#define NPF_E_BAD_CALLBACK_FUNCTION (NPF_FOUNDATIONS_BASE_ERR + 2)
#define NPF_E_RESOURCE_EXISTS (NPF_FOUNDATIONS_BASE_ERR + 3)
#define NPF_E_RESOURCE_NONEXISTENT (NPF_FOUNDATIONS_BASE_ERR + 4)

#define NPF_IPV4_BASE_ERR (NPF_FOUNDATIONS_MAX_ERR + 1)
#define NPF_IPV4_MAX_ERR (NPF_IPV4_BASE_ERR + 99)

#define NPF_INTERFACES_BASE_ERR (NPF_IPV4_MAX_ERR + 1)
#define NPF_INTERFACES_MAX_ERR (NPF_INTERFACES_BASE_ERR + 99)

#ifdef __cplusplus
}
#endif

#endif /* __NPF_H__ */
```

Appendix B List of companies belonging to NPF during approval process

Agere Systems	IBM	Samsung Electronics
Alcatel	IDT	Sandburst Corporation
Altera	Intel	Silicon & Software Systems
AMCC	IP Infusion	Silicon Access
Analog Devices	Kawasaki LSI	Sony Electronics
Avici Systems	LSI Logic	STMicroelectronics
Azanda Network Devices	Modelware	Sun Microsystems
Cypress Semiconductor	Mosaid	Teja Technologies
Ericsson	Motorola	TranSwitch
Erlang Technologies	NEC	U4EA Group
EZ Chip	NetLogic	Xelerated
Flextronics	Nokia	Xilinx
Fujitsu Ltd.	Paion Co., Ltd.	Zettacom
FutureSoft	PMC Sierra	ZTE
HCL Technologies	RadiSys	
Hi/fn		