

Process for Adding SAI Extensions

|  |  |
| --- | --- |
| **Title** | **Process for Adding SAI Extensions** |
| **Authors** | **DELL** |
| **Status** | **In Review** |
| **Type** | **Standards Track** |
| **Created** | **11/28/2016** |
| **Modified** | **07/18/2017** |
| **SAI-Version** | **V1.3** |

# SAI Extensions

SAI Extensions are used for introducing:

* Experimental ‘custom’ object attributes
* Experimental SAI modules (new SAI objects)

The goals of defining SAI extensions are to:

* Allow innovation
* Increase velocity
* Provide a formalism for publishing experimental features

## Experimental Extensions Concepts

Experimental extensions have the following characteristics:

* They are optional to SAI implementations. Vendors A and B may choose to implement the same extensions – attribute A and module M. Vendor C may choose to implement either attribute A or module M or neither.
* Must be published
* Must follow the SAI approach to define module API’s (create/get/set/delete, with key /value pairs for attributes).
* The implicit intent for experimental extensions is to eventually become part of the main SAI API – if they have enough SAI community support.
* A module or attribute should be accepted as an experimental extension if supported by at least two members of the SAI community.

## Extension (Custom) Attributes

Experimental attributes have their own range. Custom SAI attributes are defined in an extension header file associated to the module they belong to. This avoids code churn for the main SAI module definitions.

Experimental attributes must not require changes to existing SAI data structures or other existing SAI API’s. However, new data structures (if applicable) and/or enumerated types can be added to the extension header file – please refer to the usage notes.

Experimental extensions are included in the experimental directory. For instance:

**experimental/saiportextensions.h**

#include <saiport.h>

typedef enum \_sai\_port\_experimental\_attr\_t

{

…

/\*\* Custom range base value \*/

SAI\_PORT\_ATTR\_CUSTOM\_RANGE\_START /\* defined in saiport.h \*/

SAI\_PORT\_ATTR\_EXPERIMENTAL\_CAPABILITY\_X,

…

} sai\_port\_experimental\_attr\_t;

A file called sai***module***extensions.h contains enum values specific to extension attributes, e.g. saiportextensions.h. It is recommended (but not mandatory!) to use the keyword EXPERIMENTAL when defining experimental attributes – this allows developers to easily detect usage of extension attributes in source code.

There can be only one experimental sai***module***extensions.h file.

## Experimental SAI Modules

Each experimental SAI Module must define its own header file, as is the case for any regular module.

The module **must** contain a doxygen warning - to state that it is an extension, and thus may not be supported by all SAI implementations.

The module description must state what SAI implementation(s) support this module.

/\*\*

…

\* @file saiexperimentalmodule.h

\* @brief This module defines an experimental feature

\* **@description Supported by: Acme Corp, Better Electrons Inc., Crossed Wires Ltd.**

\* **@warning This module is a SAI experimental module.**

/

### SAI Module Extensions File

Definitions of new extension / experimental modules APIs are added to saiextensions.h file (rather than sai.h). The documentation of the module name must state that the module is an extension.

saiextensions.h

#include <sai.h>

typedef enum \_sai\_api\_extensions\_t

{

…

**SAI\_API\_CUSTOM\_EXPERIMENTAL\_START = SAI\_API\_CUSTOM\_RANGE\_START /\* from sai.h \*/,**

**/\*\* Experimental module. \*/**

**SAI\_API\_NEW\_PROTO\_MODULE = SAI\_API\_CUSTOM\_EXPERIMENTAL\_START + 1,**

**…**

} sai\_api\_extensions\_t;

Reasoning: use a new file to avoid code churn for sai.h.

## Testing Experimental Modules and Attributes

A test methodology and/or scripts should be defined for all experimental attributes and modules – but this is not mandatory. Formal testing only becomes mandatory when experimental attributes and modules are propagated to the main/official SAI API.

## Usage Notes for Experimental Attributes

Assume a vendor (vendor A) suggest to provide an extension – say a port attribute that controls a port LED state, and this attribute is agreed upon at least by one other SAI community member.

### Header File – Experimental Attribute

In this case, the vendor A adds an LED State attribute to the saiportextensions.h file:

**experimental/saiportextensions.h**

typedef enum \_sai\_port\_extensions\_attr\_t {

…

**SAI\_PORT\_ATTR\_LED\_STATE = SAI\_PORT\_ATTR\_CUSTOM\_RANGE\_START + 1,**

…

} sai\_port\_attr\_extensions\_t;

*// Data type for the experimental attribute LED\_STATE*

/\*\*

\* @brief Attribute data for SAI\_PORT\_ATTR\_LED\_STATE

\*/

typedef enum \_sai\_port\_led\_state\_t

{

/\*\* Unknown \*/

SAI\_PORT\_LED\_STATE\_UNKNOWN,

/\*\* ON \*/

SAI\_PORT\_LED\_ON,

/\*\* Down \*/

SAI\_PORT\_LED\_OFF,

} sai\_port\_led\_state\_t;

### Host Adapter – Experimental Attribute

…

**#include <saiport.h>**

**#include <experimental/saiportextensions.h>**

…

sai\_attribute\_t attr;

attr.id = **SAI\_PORT\_ATTR\_LED\_STATE;**

attr.u32.value = **SAI\_PORT\_LED\_ON**;

sai\_status\_t status = sai\_set\_port\_attribute(port\_id, &attr);

…

The new attribute does not impact in any way the code provided by a Vendor B which may not support that attribute. However, the Host Adapter implementation needs to be able to distinguish cases where the SAI implementation does or does not support the new experimental attribute.

Note. Explicitly including the “extensions” header file in the source code is intended. Source files which refer extensions can be easily identified, e.g. grep –r extensions . or grep –r experimental . !

## Usage Notes for Experimental Modules

SAI community members A and B may agree to define a common experimental MPLS API. This is the desired approach for mainstream protocols such as MPLS. SAI community members A and B should agree on a common experimental API which can eventually be propagated to the main SAI API.

### Header Files – Experimental Module

In this case, saiextensions.h contains:

typedef enum \_sai\_api\_extensions\_t

{

…

**/\*\* MPLS experimental extension module. \*/**

**SAI\_API\_MPLS = SAI\_API\_CUSTOM\_EXPERIMENTAL\_START + 11,**

} sai\_api\_extensions\_t;

An experimental MPLS extension header file must be defined:

**experimental/saimpls.h**

…

typedef sai\_status\_t (\*sai\_mpls\_object\_create\_fn)(\_Out\_ sai\_object\_id\_t \*mpls\_id, …);

…

typedef struct \_sai\_mpls\_api\_t {

sai\_mpls\_object\_create\_fn mpls\_obj\_create\_fn;

sai\_mpls\_object\_delete\_fn mpls\_obj\_delete\_fn;

…

} sai\_mpls\_api\_t;

…

The experimental MPLS API module must use the SAI API general approach – define create, get, set and delete functions.

### Host Adapter – Experimental Module

A host adapter implementation must include the MPLS extension module header file. Using an experimental module follows the same approach as for any other modules.

#include “experimental/saimpls.h”

…

sai\_mpls\_api\_t sai\_mpls\_api\_tbl;

sai\_status\_t status = sai\_api\_query(SAI\_API\_MPLS, (void )&sai\_mpls\_api\_tbl);

…