

|  |
| --- |
|  |

Switch Abstraction Interface

Change Proposal

|  |  |
| --- | --- |
| **Title** | **SAI Warm boot** |
| **Authors** | **DELL** |
| **Status** | **In Review** |
| **Type** | **Standards Track** |
| **Created** | **10/07/2015** |
| **SAI-Version** | **V0.9.4** |

**Contents**

[List of Changes i](#_Toc438624624)

[1 Overview 1](#_Toc438624625)

[2 Proposal 1](#_Toc438624626)

[2.1 In-Service Restart 1](#_Toc438624627)

[2.2 In-Service Upgrade 2](#_Toc438624628)

[3 Specification 2](#_Toc438624630)

[3.1 Changes to saiswitch.h 2](#_Toc438624631)

[3.2 Changes to saitypes.h 3](#_Toc438624632)

[3.3 Changes to saistatus.h 3](#_Toc438624633)

[3.4 saiobject.h (new) 4](#_Toc438624634)

[4 Configuration Example 5](#_Toc438624635)

# List of Changes

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Changes | Name | Date |
| 0.9.3 | Proposal for SAI Warm Boot – Base Version |  | 10/07/15 |

License

© 2014 Microsoft Corporation, Dell Inc., Facebook, Inc, Broadcom Corporation, Intel Corporation, Mellanox Technologies Ltd.

As of September 9, 2014, the following persons or entities have made this Specification available under the Open Web Foundation Final Specification Agreement (OWFa 1.0), which is available at <http://www.openwebfoundation.org/legal/the-owf-1-0-agreements/owfa-1-0>

Microsoft Corporation, Dell Inc., Facebook, Inc, Intel Corporation, Mellanox Technologies Ltd.

You can review the signed copies of the Open Web Foundation Agreement Version 1.0 for this Specification at <http://opencompute.org/licensing/>, which may also include additional parties to those listed above.

Your use of this Specification may be subject to other third party rights. THIS SPECIFICATION IS PROVIDED "AS IS." The contributors expressly disclaim any warranties (express, implied, or otherwise), including implied warranties of merchantability, noninfringement, fitness for a particular purpose, or title, related to the Specification. The entire risk as to implementing or otherwise using the Specification is assumed by the Specification implementer and user. IN NO EVENT WILL ANY PARTY BE LIABLE TO ANY OTHER PARTY FOR LOST PROFITS OR ANY FORM OF INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION OR ITS GOVERNING AGREEMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND WHETHER OR NOT THE OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE FOLLOWING IS A LIST OF MERELY REFERENCED TECHNOLOGY: Microprocessor technology, semiconductor manufacturing technology, operating system technology (including without limitation networking operating system technology), emulation technology, graphics technology, video technology, integrated circuit packaging technology and the like, compiler technologies, object oriented technology, optical/RF communications technology including chip I/O and driver technology, bus technology, memory chip technology (including, without limitation, NAND memory, NOR memory, resistive RAM (RRAM), seek scan probe (SSP) memory, nonvolatile memory (including without limitation, memory based on chalcogenide materials, phase change memory (PCM), one or more stacked layers of memory cells, embedded PCM memories, non-volatile cache memory, solid state drives, SRAM, embedded DRAM, ferro-electric memory, and polymer memory)) and/or health-related and medical technology. IMPLEMENTATION OF THESE TECHNOLOGIES MAY BE SUBJECT TO THEIR OWN LEGAL TERMS.

# Overview

The proposal deals with Restart and Upgrade of SAI without impacting the data plane.

**In-Service Restart**: The mechanism of restarting a component without impact to service. This assumes that the software version of the component has not changed after the restart.

**In-Service Upgrade**: The mechanism of upgrading to a newer version of a component without impacting service.

The process consists of two interactions, one between SAI and the underlying NPU SDK and the other between the Host Adapter and SAI. The proposal deals with the interaction between the Host Adapter and SAI alone.

# Proposal

SAI Init blocks till SDK reconciliation is complete

HA initiates traversal of all objects in SAI

HA may add/delete configuration to bring SAI in sync

SAI stores the data to NV store periodically during each set or after getting restart imminent event

SAI

Host Adapter

Set Request

Shutdown switch

NV Store

SAI

RESTART

SAI init switch

Get Request

**. . . . . .**

## In-Service Restart

During init SAI exchanges the below parameters required for a restart.

1. Restart type – Planned or unplanned or both. The choice depends on the capability of the underlying hardware.
2. Nonvolatile storage – Required by SAI and the NPU to store and retrieve the data stored prior to restart. The size of the storage will include the space required both by the NPU and SAI. Both SAI and NPU may decide to store the data either periodically or based on a trigger from the Host Adapter.
3. Minimum restart interval – The minimum time interval required by SAI to come out of restart.

The Host Adapter may trigger the SAI restart with shutdown\_switch(true) for a planned warm restart restart. SAI may dump its contents to the data store and the underlying NPU would also enter the restart sequence.

When host adapter re-initiates the SAI, setup init profile KV values to and NV storage location, and then call initiate\_switch(). During this process SAI Init will block and Host Adapter should wait for the minimum restart interval before bringing down SAI due to init failure. When warm boot fail, host adapter can set warm boot type to 0, and call initiate\_switch() to do cold boot.

Once init is complete the Host Adapter will choose to retrieve the SAI objects to bring the Host Adapter in sync with the SAI objects. This would be done by Host Adapter and SAI would be passive.

Unplanned restart support depends on the underlying NPU and SAI will provide best effort reconciliation with the NPU. A failure would be declared if unplanned restart is not supported.

## In-Service Upgrade

The upgrade process would follow the planned restart sequence with the difference that SAI would come up with a new software version after restart. In case of incompatible version the Host Adapter would bring down SAI and log a critical error.

# Specification

## Changes to saiswitch.h

typedef enum \_sai\_switch\_restart\_type\_t

{

/\* Planned restart only \*/

SAI\_RESTART\_TYPE\_PLANNED,

/\* Unplanned restart only \*/

SAI\_RESTART\_TYPE\_UNPLANNED,

/\* Both planned and unplanned restart \*/

SAI\_RESTART\_TYPE\_BOTH,

/\* NPU doesn’t support warmboot \*/

SAI\_RESTART\_TYPE\_NONE,

} sai\_switch\_restart\_type\_t;

typedef enum \_sai\_switch\_attr\_t

{

/\* READ\_ONLY \*/

/\* Type of restart supported [sai\_switch\_restart\_type\_t].\*/

SAI\_SWITCH\_ATTR\_RESTART\_TYPE,

/\* READ\_ONLY \*/

/\* Minimum interval of time required by SAI for planned restart [sai\_uint32\_t]

in milliseconds. Will be 0 for SAI\_RESTART\_TYPE\_NONE \*/

/\* The Host Adapter will have to wait for this minimum interval of time before it decides

to bring down SAI due to init failure. \*/

SAI\_SWITCH\_ATTR\_MIN\_PLANNED\_RESTART\_INTERVAL,

/\* READ\_ONLY \*/

/\* Nonvolatile storage required by both SAI and NPU [sai\_uint64\_t]. Will be 0

for SAI\_RESTART\_TYPE\_NONE \*/

SAI\_SWITCH\_ATTR\_NV\_STORAGE\_SIZE,

} sai\_switch\_attr\_t;

**Key value pair for reboot type and filename used for recovery**

/\* 0: means cold boot, and 1: means warm boot \*/

#define SAI\_KEY\_WARM\_BOOT “SAI\_WARM\_BOOT”

/\* The file to recover SAI/NPU state from \*/

#define SAI\_KEY\_WARM\_BOOT\_READ\_FILE “SAI\_WARM\_BOOT\_READ\_FILE”

/\* The file to write SAI/NPU state to \*/

#define SAI\_KEY\_WARM\_BOOT\_WRITE\_FILE “SAI\_WARM\_BOOT\_WRITE\_FILE”

## Changes to saitypes.h

typedef enum \_sai\_object\_type\_t {

SAI\_OBJECT\_TYPE\_FDB = 26,

SAI\_OBJECT\_TYPE\_ROUTE = 27,

SAI\_OBJECT\_TYPE\_NEIGHBOR = 28,

SAI\_OBJECT\_TYPE\_VLAN = 29,

SAI\_OBJECT\_TYPE\_MAX = 30,

} sai\_object\_type\_t;

typedef struct \_sai\_object\_key\_t

{

union

{

sai\_object\_id\_t object\_id;

sai\_vlan\_id\_t vlan\_id;

sai\_fdb\_entry\_t fdb\_entry;

sai\_neighbor\_entry\_t neighbor\_entry;

sai\_unicast\_route\_entry\_t route\_entry;

} key;

} sai\_object\_key\_t;

## Changes to saistatus.h

/\* The NV storage used is either invalid or corrupt. (rv for initialize\_switch()) \*/

SAI\_STATUS\_INVALID\_NV\_STORAGE SAI\_STATUS\_CODE(0x00000014L)

(rv for initialize\_switch())

/\* The NV storage is full. (rv for shutdown\_switch()) \*/

SAI\_STATUS\_NV\_STORAGE\_FULL SAI\_STATUS\_CODE(0x00000015L)

## saiobject.h (new)

The bulk api uses the sai\_object\_key\_t structure. A new header file saiobject.h is defined to include the bulk API without modifying each of the headers.

/\*

\* Routine Description:

\* @brief Get maximum number of attributes for an object type

\* Arguments:

\* [in] sai\_object\_type\_t - SAI object type

\* [Out] count – maximum number of attribute for an object type

\*

\* Return Values:

\* SAI\_STATUS\_SUCCESS on success

\* Failure status code on error

\*/

sai\_status\_t sai\_get\_maximum\_attribute\_count(

\_In\_ sai\_object\_type\_t object\_type,

\_Out\_ uint32\_t \*count);

/\*

\* Routine Description:

\* @brief Get the number of objects present in SAI

\* Arguments:

\* [in] sai\_object\_type\_t - SAI object type

\* [inout] count – number of objects in SAI

\*

\* Return Values:

\* SAI\_STATUS\_SUCCESS on success

\* Failure status code on error

\*/

sai\_status\_t sai\_get\_object\_count(

\_In\_ sai\_object\_type\_t object\_type,

\_InOut\_ uint32\_t \*count);

/\*

\* Routine Description:

\* @brief Get the list of object keys present in SAI

\* Arguments:

\* [in] sai\_object\_type\_t - SAI object type

\* [in] count – number of objects in SAI

\* [in] object\_list – List of SAI objects or keys

\* Return Values:

\* SAI\_STATUS\_SUCCESS on success

\* Failure status code on error

\*/

sai\_status\_t sai\_get\_object\_key(

\_In\_ sai\_object\_type\_t object\_type,

\_In\_ uint32\_t object\_count,

\_InOut\_ sai\_object\_key\_t \*object\_list);

/\*

\* Routine Description:

\* @brief Get the bulk list of valid attributes for a given list of object keys. Only valid attributes for an objects are returned.

\* Arguments:

\* [in] object\_type – sai object type

\* [in] object\_count – number of objects

\* [in] object\_key - List of object keys

\* [inout] attr\_count – List of attr\_count. Caller passes the number of attribute allocated in. Callee returns with the actual number of attributes filled in. If the count is less than needed, callee fills with the needed count and do not fill the attributes. Callee also set the corresponding status to SAI\_STATUS\_BUFFER\_OVERFLOW.

\* [inout] attrs – list of attributes for every object. Caller is responsible for allocating and freeing buffer for the attributes. For list based attribute, e.g., s32list, oidlist, callee should assume the caller has not allocate the memory for the list and should only to fill the count but not list. Then, calller can use corresponding get\_attribute to get the list.

\* [inout] object\_statuses – status for each object. If the object does not exist, callee sets the correpsonding status to SAI\_STATUS\_INVALID\_OBJECT\_ID. If the allocated attribute count is not large enough, set the status to SAI\_STATUS\_BUFFER\_OVERFLOW.

\* Return Values:

\* SAI\_STATUS\_SUCCESS on success

\* Failure status code on error

\*\*/

sai\_status\_t sai\_bulk\_get\_attribute(

\_In\_ sai\_object\_type\_t object\_type,

\_In\_ uint32\_t object\_count,

\_In\_ sai\_object\_key\_t \*object\_key,

\_Inout\_ uint32\_t \*attr\_count,

\_Inout\_ sai\_attribute\_t \*\*attrs,

\_Inout sai\_status\_t \*object\_statuses);

# Configuration Example

uint32\_t restart\_type;

uint64\_t nv\_storage\_required;

uint32\_t object\_count = 0;

uint32\_t attr\_count = 1;

uint32\_t object\_count = 0;

bool planned\_restart = true;

sai\_status\_t \*statuses;

sai\_object\_id\_t \*object\_list = NULL ;

sai\_attr\_id\_t \*attr\_id = NULL ;

sai\_attr\_t attr;

/\* Get the restart type \*/

attr.id = SAI\_SWITCH\_ATTR\_RESTART\_TYPE ;

sai\_get\_switch\_attribute\_fn(attr\_count, &attr);

restart\_type = attr.value.u32;

/\* Get the NV storage size required. \*/

attr.id = SAI\_SWITCH\_ATTR\_NV\_STORAGE\_SIZE;

sai\_get\_switch\_attribute\_fn(attr\_count, &attr);

nv\_storage\_required = attr.value.u64;

/\* Trigger warmboot by calling switch shutdown function \*/

/\* SAI stores its contents to the file. The NPU also stores its contents to the file.\*/

sai\_shutdown\_switch\_fn(planned\_restart);

/\* After SAI comes from reboot Host Adapter uses the Key/Value pair

to retrieve the restart type and filename used for storage \*/

/\* Based on the restart type SAI does a cold boot or warmboot and reconciles its contents

from the file \*/

PROFILE[SAI\_KEY\_WARM\_BOOT] = “1”

PROFILE[SAI\_KEY\_WARM\_BOOT\_READ\_FILE] = “/var/cache/sai\_warmboot.bin”

PROFILE[SAI\_KEY\_WARM\_BOOT\_WRITE\_FILE] = “/var/cache/sai\_warmboot.bin”

sai\_switch\_api->initialize\_switch();

/\* The Host Adapter will query the object count, object list and attributes for each object

to compare its contents and bring SAI to the same state as Host Adapter. \*/

sai\_get\_object\_count(SAI\_OBJECT\_TYPE\_LAG, &object\_count);

object\_keys = (sai\_object\_key\_t \*)calloc(object\_count, sizeof(sai\_object\_key\_t));

sai\_get\_object\_key(SAI\_OBJECT\_TYPE\_LAG, object\_count, object\_keys);

sai\_get\_maximum\_attribute\_count(SAI\_OBJECT\_TYPE\_LAG, &attr\_count);

sai\_uint32\_t attrs\_count[object\_count];

for (i = 0; i < object\_count; ++i)  
{

attrs\_count[i] = attr\_count;

}

sai\_attribute\_t attrs[object\_count][attr\_count];

statuses = (sai\_status\_t \*)calloc(object\_count, sizeof(sai\_status\_t));

sai\_bulk\_get\_attribute\_fn(SAI\_OBJECT\_TYPE\_LAG, object\_count, object\_keys, attrs\_count,

attrs, statuses);

for (i = 0; i < object\_count; ++i)

{

for (j = 0; < attrs\_count[j]; ++j)

{

if (attrs[i][j].id == SAI\_LAG\_ATTR\_PORT\_LIST)

{

port\_count = attrs[i][j].value.oidlist.count;

sai\_attribute\_t attr;

attr.id = SAI\_LAG\_ATTR\_PORT\_LIST;

attr.value.oidlist.list = new sai\_object\_id\_t[port\_count];

sai\_lag\_api->get\_attribute(object\_keys[i], 1, attr);

free(attr.value.oidlist.list)

}

}

}

/**\* In case of FDB, ROUTE, VLAN, ROUTE which don’t have objects \*/**

sai\_fdb\_entry\_t \*fdb\_entry\_list = NULL;

sai\_get\_object\_count(SAI\_OBJECT\_TYPE\_FDB, &object\_count);

fdb\_entry\_list = (sai\_object\_id\_t \*)calloc(object\_count, sizeof(sai\_object\_key\_t));

sai\_get\_object\_key\_fn(SAI\_OBJECT\_TYPE\_FDB, object\_count, fdb\_entry\_list);

sai\_get\_maximum\_attribute\_count(SAI\_OBJECT\_TYPE\_FDB, &attr\_count);

sai\_uint32\_t attrs\_count[object\_count];

for (i = 0; i < object\_count; ++i)  
{

attrs\_count[i] = attr\_count;

}

sai\_attribute\_t attrs[object\_count][attr\_count];

statuses = (sai\_status\_t \*)calloc(object\_count, sizeof(sai\_status\_t));

sai\_bulk\_get\_attribute(SAI\_OBJECT\_TYPE\_FDB, object\_count, fdb\_entry\_list,

attrs\_count, attrs, statuses);