Service Availability™ Forum Application Interface Specification

Notification Service SAI-AIS-NTF-A.01.01

The Service Availability™ solution is high availability and more; it is the delivery of ultra-dependable communication services on demand and without interruption.

This Service Availability™ Forum Application Interface Specification document might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Current characterized errata are available on request.

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

The Service Availability™ Specification(s) (the "Specification") found at the URL http://www.saforum.org (the "Site") is generally made available by the Service Availability Forum (the "Licensor") for use in developing products that are compatible with the standards provided in the Specification. The terms and conditions, which govern the use of the Specification are set forth in this agreement (this "Agreement").

IMPORTANT – PLEASE READ THE TERMS AND CONDITIONS PROVIDED IN THIS AGREEMENT BEFORE DOWNLOADING OR COPYING THE SPECIFICATION. IF YOU AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, CLICK ON THE "ACCEPT" BUTTON. BY DOING SO, YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS STATED IN THIS AGREEMENT. IF YOU DO NOT WISH TO AGREE TO THESE TERMS AND CONDITIONS, YOU SHOULD PRESS THE "CANCEL"BUTTON AND THE DOWNLOAD PROCESS WILL NOT PROCEED.

1. LICENSE GRANT. Subject to the terms and conditions of this Agreement, Licensor hereby grants you a non-exclusive, worldwide, non-transferable, revocable, but only for breach of a material term of the license granted in this section 1, fully paid-up, and royalty free license to:

a. reproduce copies of the Specification to the extent necessary to study and understand the Specification and to use the Specification to create products that are intended to be compatible with the Specification;

b. distribute copies of the Specification to your fellow employees who are working on a project or product development for which this Specification is useful; and

c. distribute portions of the Specification as part of your own documentation for a product you have built, which is intended to comply with the Specification.

2. DISTRIBUTION. If you are distributing any portion of the Specification in accordance with Section 1(c), your documentation must clearly and conspicuously include the following statements:

a. Title to and ownership of the Specification (and any portion thereof) remain with Service Availability Forum ("SA Forum").

b. The Specification is provided "As Is." SA Forum makes no warranties, including any implied warranties, regarding the Specification (and any portion thereof) by Licensor.

c. SA Forum shall not be liable for any direct, consequential, special, or indirect damages (including, without limitation, lost profits) arising from or relating to the Specification (or any portion thereof).

d. The terms and conditions for use of the Specification are provided on the SA Forum website.

3. RESTRICTION. Except as expressly permitted under Section 1, you may not (a) modify, adapt, alter, translate, or create derivative works of the Specification, (b) combine the Specification (or any portion thereof) with another document, (c) sublicense, lease, rent, loan, distribute, or otherwise transfer the Specification to any third party, or (d) copy the Specification for any purpose.

4. NO OTHER LICENSE. Except as expressly set forth in this Agreement, no license or right is granted to you, by implication, estoppel, or otherwise, under any patents, copyrights, trade secrets, or other intellectual property by virtue of your entering into this Agreement, downloading the Specification, using the Specification, or building products complying with the Specification.

5. OWNERSHIP OF SPECIFICATION AND COPYRIGHTS. The Specification and all worldwide copyrights therein are the exclusive property of Licensor. You may not remove, obscure, or alter any copyright or other proprietary rights notices that are in or on the copy of the Specification you download. You must reproduce all such notices on all copies of the Specification you make. Licensor may make changes to the Specification, or to items referenced

therein, at any time without notice. Licensor is not obligated to support or update the Specification.

6. WARRANTY DISCLAIMER. THE SPECIFICATION IS PROVIDED "AS IS." LICENSOR DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT OF THIRD-PARTY RIGHTS, FITNESS FOR ANY PARTICULAR PURPOSE, OR TITLE. Without limiting the generality of the foregoing, nothing in this Agreement will be construed as giving rise to a warranty or representation by Licensor that implementation of the Specification will not infringe the intellectual property rights of others.

7. PATENTS. Members of the Service Availability Forum and other third parties [may] have patents relating to the Specification or a particular implementation of the Specification. You may need to obtain a license to some or all of these patents in order to implement the Specification. You are responsible for determining whether any such license is necessary for your implementation of the Specification and for obtaining such license, if necessary. [Licensor does not have the authority to grant any such license.] No such license is granted under this Agreement.

8. LIMITATION OF LIABILITY. To the maximum extent allowed under applicable law, **LICENSOR DISCLAIMS ALL LIABILITY AND DAMAGES, INCLUDING DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, AND INCIDENTAL DAMAGES, ARISING FROM OR RELATING TO THIS AGREEMENT, THE USE OF THE SPECIFICATION OR ANY PRODUCT MANUFACTURED IN ACCORDANCE WITH THE SPECIFICATION, WHETHER BASED ON CONTRACT, ESTOPPEL, TORT, NEGLIGENCE, STRICT LIABILITY, OR OTHER THEORY. NOTWITHSTANDING ANYTHING TO THE CONTRARY, LICENSOR'S TOTAL LIABILITY TO YOU ARISING FROM OR RELATING TO THIS AGREEMENT OR THE USE OF THE SPECIFICATION OR ANY PRODUCT MANUFACTURED IN ACCORDANCE WITH THE SPECIFICATION WILL NOT EXCEED ONE HUNDRED DOLLARS ($100). YOU UNDERSTAND AND AGREE THAT LICENSOR IS PROVIDING THE SPECIFICATION TO YOU AT NO CHARGE AND, ACCORDINGLY, THIS LIMITATION OF LICENSOR'S LIABILITY IS FAIR, REASONABLE, AND AN ESSENTIAL TERM OF THIS AGREEMENT.**

9. TERMINATION OF THIS AGREEMENT. Licensor may terminate this Agreement, effective immediately upon written notice to you, if you commit a material breach of this Agreement and do not cure the breach within ten (30) days after receiving written notice thereof from Licensor. Upon termination, you will immediately cease all use of the Specification and, at Licensor's option, destroy or return to Licensor all copies of the Specification and certify in writing that all copies of the Specification have been returned or destroyed. Parts of the Specification that are included in your product documentation pursuant to Section 1 prior to the termination date will be exempt from this return or destruction requirement.

10. ASSIGNMENT. You may not assign, delegate, or otherwise transfer any right or obligation under this Agreement to any third party without the prior written consent of Licensor. Any purported assignment, delegation, or transfer without such consent will be null and void.

11. GENERAL. This Agreement will be construed in accordance with, and governed in all respects by, the laws of the State of Delaware (without giving effect to principles of conflicts of law that would require the application of the laws of any other state). You acknowledge that the Specification comprises proprietary information of Licensor and that any actual or threatened breach of Section 1 or 3 will constitute immediate, irreparable harm to Licensor for which monetary damages would be an inadequate remedy, and that injunctive relief is an appropriate remedy for such breach. All waivers must be in writing and signed by an authorized representative of the party to be charged. Any waiver or failure to enforce any provision of this Agreement on one occasion will not be deemed a waiver of any other provision or of such provision on any other occasion. This Agreement may be amended only by binding written instrument signed by both parties. This Agreement sets forth the entire understanding of the parties relating to the subject matter hereof and thereof and supersede all prior and contemporaneous agreements, communications, and understandings between the parties relating to such subject matter.

Table of Contents Notification Service

1 Document Introduction

1.1 Document Purpose

This document defines the Notification Service of the Application Interface Specification (AIS) of the Service Availability™ Forum. It is intended for use by implementers of the Application Interface Specification and Hardware Platform Interface (HPI) as well as application developers who use the Application Interface Specification to develop applications. The specification is defined in the C programming language, and requires substantial knowledge of the C programming language.

Typically, the Service Availability™ Forum Application Interface Specification will be used in conjunction with the Service Availability™ Forum Hardware Platform Interface (HPI) and with the Service Availability™ Forum System Management Specification (SMS).

1.2 AIS Documents Organization

The Application Interface Specification is organized into several volumes. For a list of all Application Interface Specification documents, refer to the SA Forum Overview document [5].

1.3 History

SAI-AIS-NTF-A.01.01 is the first release of the NTF Service specification.

1.4 References

The following documents contain information that is relevant to this specification:

[1] CCITT Recommendation X.730 | ISO/IEC 10164-1, Object Management Function

[2] CCITT Recommendation X.731 | ISO/IEC 10164-2, State Management Function

[3] CCITT Recommendation X.733 | ISO/IEC 10164-4, Alarm Reporting Function

[4] CCITT Recommendation X.736 | ISO/IEC 10164-7, Security Alarm Reporting Function

[5] Service Availability™ Forum, Application Interface Specification, Overview, SAI-Overview-B.02.01

[6] Service Availability™ Forum, Application Interface Specification, Event Service, SAI-AIS-EVT-B.02.01

[7] SNMP enterprise numbers, http://www.iana.org/assignments/enterprise-numbers

References to these documents are made by putting the number of the document in brackets.

1.5 How to Provide Feedback on the Specification

If you have a question or comment about this specification, you may submit feedback online by following the links provided for this purpose on the Service Availability™ Forum website (**http://www.saforum.org**).

You can also sign up to receive information updates on the Forum or the Specification.

1.6 How to Join the Service Availability™ Forum

The Promoter Members of the Forum require that all organizations wishing to participate in the Forum complete a membership application. Once completed, a representative of the Service Availability™ Forum will contact you to discuss your membership in the Forum. The Service Availability™ Forum Membership Application can be completed online by following the pertinent links provided on the Forum's website (**http://www.saforum.org**).

You can also submit information requests online. Information requests are generally responded to within three business days.

1.7 Additional Information

1.7.1 Member Companies

A list of the Service Availability™ Forum member companies can also be viewed online by using the links provided on the Forum's website (**http://www.saforum.org**).

1.7.2 Press Materials

The Service Availability™ Forum has available a variety of downloadable resource materials, including the Forum Press Kit, graphics, and press contact information.

Visit this area often for the latest press releases from the Service Availability™ Forum and its member companies by following the pertinent links provided on the Forum's website (**http://www.saforum.org**).

2 Overview

ITU-T recommendations X.700 - X.799 deal with the area of system management and how it may be applied to a communications system. ITU-T broadly classifies the management domain into the famous FCAPS model that segregates the overall management into five areas, with the "F" standing for Fault Management. The Notification service is based on these Fault management recommendations to a great degree, but also needs many other supportive recommendations that include, for example, the concepts of Managed objects, which are covered in Structure of Management Information. There are also normative references to ITU-T defined agents and managers that are extensively used in the definition of the current notification standards.

Adapting the definition from ITU-T X.710 to the present SA Forum context, lead to the following definition:

The Notification Service is used by a service-user to report an event to a peer service-user. It is defined as a non-confirmed service.

Event here means the same as in commonly understood English - an incident, or simply, a change of status.

Note: In order to avoid confusion with the Event Service (which is specified in [6]), this document instead uses the term *notification*.

The entities related to the Notification Service are shown in the following figure.

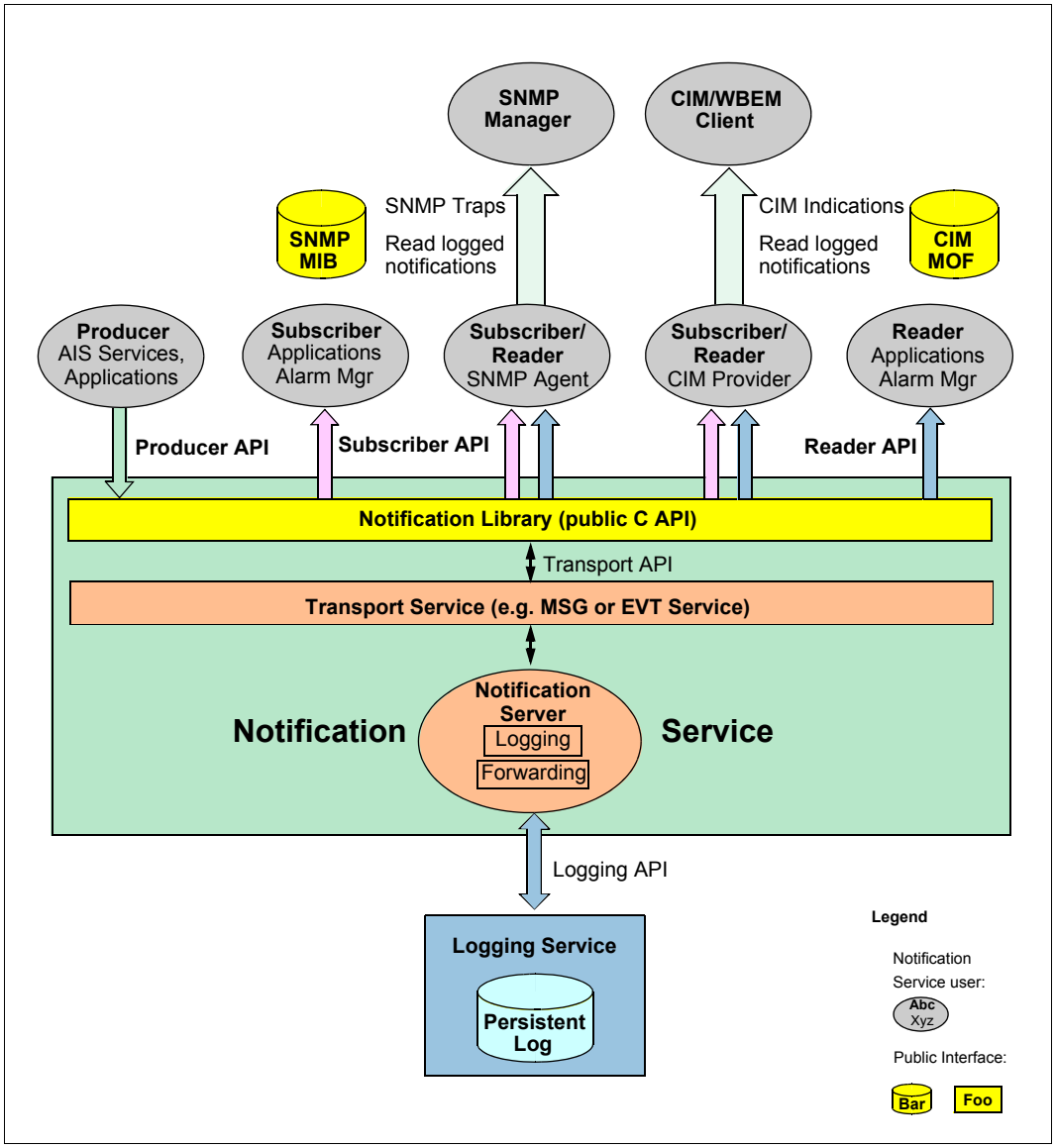

Figure 1: Notification Entities

2.1 Users of the Notification Library

Users of the Notification Library run on the nodes of a SA Forum cluster.

2.1.1 Producer

A notification producer generates notifications (using the producer API of the Notification Service).

2.1.2 Consumer

A consumer consumes notifications that were generated by producers. If not interested in all notifications, filter criteria can be specified. A consumer can be one of the following types or both. A consumer can also be a producer.

2.1.2.1 Subscriber

A subscriber for notifications gets notifications forwarded as they occur (push interface).

2.1.2.2 Reader

A reader retrieves historical notification entries from the persistent notification log (pull interface).

2.2 SNMP Interface

The Notification MIB is currently not defined but may be defined in the future. Examples for possible functionality via SNMP by the Notification MIB are:

- Forward notifications of the Notification Service as SNMP traps (or SNMP informs)
- Read logged notifications

The MIBs for the AIS services and AMF provide the schemas that allow for the following functionality via SNMP:

- Forward notifications of the AIS service and AMF as SNMP traps

The SNMP agents implementing these MIBs are not part of the Notification Service. They are users of the Notification Library and behave like subscribers and readers.

Note: The MIBs of the AIS services and AMF are not part of this document.

2.3 CIM/WBEM Interface

The Notification MOF as well as the MOFs for the AIS services and AMF are currently not defined but may be defined in the future. Examples for possible functionality via CIM/WBEM by these MOFs are:

- Forward notifications of the Notification Service as CIM indications
- Read logged notifications
- Forward notifications of the AIS service and AMF as CIM indications

2.4 Notification Service

Similar to the AIS services, the Notification Service mainly consists of a client library and a server. There are no assumptions made as to how server instances are distributed across the nodes of a SA Forum cluster. In an implementation the server even could be part of the library.

2.4.1 Notification Library

The Notification Library provides the following public C APIs:

- Producer API
- Subscriber API
- Reader API

2.4.2 Notification Server

The server applies the filtering criteria on notifications for delivery to subscribed consumers and performs the logging into persistent storage.

2.4.3 Transport Service

The Transport Service links the Notification Library and the Notification Server. It is currently not specified.

2.5 Logging Service

Alarm notifications and security alarm notifications are logged persistently. An implementation may also support persistent logging for the other notification types (object creation / deletion, attribute value change, state change notifications). It is recommended that the Notification Service use the API of the SA Forum Logging Service to write notifications into persistent storage. Likewise, the Notification Reader API may also use the log files of the Logging Service.

3 Notification Service API

3.1 Notifications

Notifications are those data objects that are generated using the Producer API. The same objects are forwarded to users of the Subscriber API and returned to users of the Reader APIs. Notifications have attributes that are closely related to those specified in the ITU-T recommendations. The notification attributes are described in subsequent sections.

3.2 Notification Filters

Notification filters are used with the Subscriber and Reader API. Their purpose is to reduce the number of notifications that are returned by these APIs and to allow a user application to specify the notifications in which it is interested. Notification filters have a subset of the attributes specified for notifications.

3.3 Notification Types

As seen earlier, notification (or event) means an incident, or simply, a change of status. Notifications are grouped into notification types. The following types of notifications can be produced and consumed in a SA Forum cluster:

- *Alarm*
- *State Change*
- *Object Create / Delete*
- Attribute Change
- *Security alarm*

3.3.1 Alarm Notification

An alarm report is a notification of a specific event that may or may not represent an error. This is defined in ITU X.733 [3].

In the context of SA Forum, AIS services, AMF, applications, HPI listener and proxies for non-SA applications can send alarm notifications. An application detecting a communication failure, operating system reaching some threshold of maximum number of files opened, AIS services or AMF running into internal errors are some examples of alarm notifications.

3.3.2 State Change Notification

A state change report is a notification to report change of state of a managed object that results through either the internal operation of the managed object or via management operation. This is defined in X.731 [2].

AMF would send such notifications in the SA Forum context. The changes of Presence state, Readiness state, HA state etc. of service units and components performed by AMF can be reported to management applications using these notifications.

3.3.3 Object Create / Delete and Attribute Change Notifications

These notifications report creation and deletion of managed objects and attribute changes of configuration data on a managed object. This is defined in X.730 [1].

AMF instantiates and terminates service units and components in the cluster. Also, AMF configuration data may be changed. AMF can report such events to management applications (including Subscribers to the Notification Service) using this type of notifications. Application-specific information like threads created, users added/ deleted, modifications to default configuration data etc. can also be reported to management applications using this notification.

3.3.4 Security Alarm Notification

This notification is used to report an event indicating an attack or potential attack on system security has been detected. This is defined in X.736 [4].

Applications would be the primary generators of this notification. Repeated login attempt failures, occurrence of an event at unexpected or prohibited time, illegal modification of data are some examples.

3.4 Common Parameters

This section describes the common parameters that are included in a notification.

Name	X.73x recommendation	Default Value
Event Type	Mandatory Parameter	–
Notification Object	Mandatory Parameter	–
Notifying Object	–	Notification Object
Notification Class Identifier	–	–
Event Time	Mandatory Parameter	–

Name	X.73x recommendation	Default Value
Notification Identifier	Optional Parameter	–
Correlated Notifications	Optional Parameter	NULL, shows this is either a first time occurrence, or that no correlation need be applied.
Additional Text	Optional Parameter	–
Additional Information	Optional Parameter	–

3.4.1 Event Type

Event type is different for each type of notifications.

For Alarm notifications, this field broadly classifies the type of error encountered as:

- Communication link failure
- QOS alarm: degradation in QOS
- Processing alarm: software or processing fault
- Equipment alarm: caused by an equipment fault
- Environment alarm: conditions of the enclosure.

For state change notifications, event type can only be a state change event.

Possible values of event type for object change notification are:

- Object creation event
- Object deletion event
- Attribute addition event
- Attribute deletion event
- Attribute change event
- Attribute reset to default event.

For security alarm notifications, the possible event types are:

- Integrity violation: information may have been illegally modified, inserted or deleted
- Operational violation: unavailability, malfunction or incorrect invocation of a service
- Physical violation: violation of a physical managed object
- Security service violation: security service has detected a security attack
- Time domain violation: an event has occurred at an unexpected or prohibited time.

It is possible to derive the Notification Type from the Event Type parameter.

3.4.2 Notification Object

This is a logical entity **about** which the notification is generated, identified by its LDAP DN.

3.4.3 Notifying Object

This is a logical entity that is sending the notification, identified by its LDAP DN.

3.4.4 Notification Class Identifier

The notification class identifier has been introduced for users of the Subscriber and Reader APIs in order to have a single (filter) criterion for uniquely identifying classes of similar notifications.

Notifications that are issued at runtime can be grouped into notification classes (NCs), where each class contains notifications for similar situations with certain varying parameter values. In other words, notifications are runtime instances of NCs. (Note that the term *class* here is used only to group runtime notifications dealing with the same kind of situation. In particular, *class* does **not** imply an inheritance mechanism as in object oriented programming languages.)

These are two examples for NCs, taken from the AIS:

- Message Service: Destination message queue <name> full.
- AMF: The HA state of SI <name> assigned to SU <name> changed.

In the first example, the above mentioned varying parameter values could be the notification object, i.e., the message queue name. In the second example, the varying parameter values could be the notification object (service instance name) and the attribute list (service unit name). Note that although the parameter values may differ for instances of these notification classes, they will always be of the same kind. In the first example, for instance, the notification object parameter will always contain a message queue name and in the second example the first parameter will always be a SI name and the second parameter will always be a SU name.

For identification purposes, each NC is assigned a unique numeric notification class identifier (NCI). In order to avoid conflicts between identifier values from different vendors and applications, the NCI is explicitly divided into a vendor identifier and a vendor specific part.

3.4.5 Notification Identifier

This parameter, when present, provides an identifier for a specific notification instance, which may be carried in the Correlated notifications parameter of future notifications. Notification identifiers must be chosen to be unique across all notifications of a particular managed object throughout the time that correlation is significant.

In a SA Forum system, this parameter shall be unique within the cluster, but it need not be cluster-wide monotonically increasing. This parameter shall be an OUT parameter i.e., the logical entity sending the notification shall not be burdened with the generation of cluster-wide unique identifier.

3.4.6 Correlated Notifications

This is a set of notifications generated earlier and that is related to this notification. Management applications can use this field as a hint for identifying all the notifications that may have caused this notification or all the notifications that may be modified (say cleared) due to this notification.

In SA Forum system, this parameter shall be able to carry none, one or more notification identifiers.

3.4.7 Event Time

This field contains the time at which an event is detected and this may not be same as the time at which it is reported.

3.4.8 Additional Text

This field allows a free form text description to be reported.

3.4.9 Additional Information

This is a data structure that may carry more data not covered by the standard fields in the report. This parameter is opaque and is intended to carry Producer-Consumer specific parameters.

3.5 Notification-specific Parameters

This section describes the additional parameters that are specific to each type of notification.

3.5.1 Alarm

Specific parameters for this report are:

Name	X.73x recommendation	Default Value
Probable Cause	Mandatory Parameter	–
Specific Problems	Optional Parameter	–
Perceived Severity	Mandatory Parameter	“Major”
Trend Indication	Optional Parameter	“No Change”
Threshold Information	Optional Parameter	–
Monitored Attributes	Optional Parameter	–
Proposed Repair Actions	Optional Parameter	–

3.5.1.1 Probable Cause

This parameter augments the information provided by the event type field and further qualifies the actual cause of alarms. Probable cause is a behavioral aspect of the logical entity and most specific probable causes shall be chosen for a logical entity. A list of generic probable causes is given in the X.733 standard.

3.5.1.2 Specific Problems

This is a further refinement to the probable cause field.

3.5.1.3 Perceived Severity

This is the severity of the notification, as seen by the entity reporting it. Six levels of severity are defined:

- *Cleared:* This means a previously reported alarm is cleared. Clearing of alarms can be done based on matching event types, probable cause and specific problem. Or it may be based on the parameters in correlated notifications.
- *Indeterminate:* Severity cannot be determined by the reporting entity.
- *Critical:* A service affecting condition
- *Major:* An urgent corrective action is required to avoid a service affecting condition

- *Minor:* A non-service affecting condition. But corrective actions are needed to avoid more problems
- *Warning:* A potential service affecting condition, before any significant effects are felt

3.5.1.4 Trend Indication

Trend indication is important when a logical entity already has outstanding alarms and more alarms are reported on the same logical entity. The trend indication field indicates whether the severity of the logical entity error is getting worse, remaining the same, or improving. This field is useful for notification filtering based on severity.

3.5.1.5 Threshold Information

If the alarm is based on a parameter exceeding a threshold, this field may be utilized to capture that information. Threshold information encapsulates the threshold identifier, the actual threshold value, threshold hysteresis (important to avoid repeated alarms), observed value of the parameter and time of last threshold crossing.

3.5.1.6 Monitored Attributes

This field is useful in reporting any changing attributes of the logical entity, which may be of interest in relation to this alarm. This field uses the same syntax as the attribute list in object creation/deletion notifications (Attribute List).

3.5.1.7 Proposed Repair Actions

If the cause of the alarm is known, one or more repair actions may be proposed using this field.

3.5.2 State Change

Specific parameters for this report are:

Name	X.73x recommendation	Default Value
Source Indicator	Optional Parameter	AMF
Changed State Attribute List	Mandatory Parameter	–
Attribute Identifier	Mandatory Parameter	–
Old Attribute Value	Optional Parameter	–
New Attribute Value	Mandatory Parameter	–

3.5.2.1 Source Indicator

This indicates whether the state change was initiated by an internal operation of the logical entity, by a management operation or by an unknown source.

3.5.2.2 Changed State Attribute List

This is a list of attribute identifiers. Multiple types of state changes (for instance, Lifecycle, Readiness, HA) can be carried in this list. However, multiple state changes of the same type within one notification are not supported.

3.5.2.2.1 Attribute Identifier

This is an identifier for the state attribute that is being modified, for instance, Lifecycle, Readiness, HA.

3.5.2.2.2 Old Attribute Value

This is the value of the state attribute before the change.

3.5.2.2.3 New Attribute Value

This is the value of the state attribute after the change.

3.5.3 Object Creation/Deletion

Specific parameters for this report are:

Name	X.73x recommendation	Default Value
Source Indicator	Optional Parameter	–
Attribute List	Optional Parameter	–
Attribute Identifier	Optional Parameter	–
Attribute Value	Optional Parameter	–

3.5.3.1 Source Indicator

This field is the same as Source Indicator.

3.5.3.2 Attribute List

This parameter is a set of attributes and their current values at the time the logical entity was created or deleted.

3.5.3.2.1 Attribute Identifier

This is an identifier for an attribute.

3.5.3.2.2 Attribute Value

This is the value of the attribute at the time of creation/ deletion.

3.5.4 Attribute Value Change

Specific parameters for this report are:

Name	X.73x recommendation	Default Value
Source Indicator	Optional Parameter	–
Changed Attribute List	Mandatory Parameter	–
Attribute Identifier	Mandatory Parameter	–
Old Attribute Value	Optional Parameter	–
New Attribute Value	Mandatory Parameter	–

3.5.4.1 Source Indicator

This field is the same as Source Indicator.

3.5.4.2 Changed Attribute List

This is a list of changed attributes. Multiple attributes can be carried in this list. However, multiple values of same attribute shall not be supported.

3.5.4.2.1 Attribute Identifier

This is an identifier for the attribute that is being modified.

3.5.4.2.2 Old Attribute Value

This is the value of the attribute before the change.

3.5.4.2.3 New Attribute Value

This is the value of the attribute after the change.

3.5.5 Security Alarm

Specific parameters for this report are:

Name	X.73x recommendation	Default Value
Cause	Mandatory Parameter	–
Severity	Mandatory Parameter	–
Detector	Mandatory Parameter	–
User	Mandatory Parameter	–
Provider	Mandatory Parameter	–

3.5.5.1 Security Alarm Cause

This field is similar to the Probable Cause field in alarm notifications (3.5.1.1 on page 22). A list of generic severity alarm causes is given in the X.736 standard.

3.5.5.2 Security Alarm Severity

Same as the severity field in alarm notifications (Perceived Severity).

3.5.5.3 Security Alarm Detector

This field indicates the detector of this security alarm.

3.5.5.4 Service User

The service user whose request for service led to the generation of this security alarm is indicated in this field.

3.5.5.5 Service Provider

The intended service provider of the service, which led to this security alarm, is indicated in this field.

3.6 Notification Delivery Characteristics

The following are the delivery characteristics for notifications generated by producers to all subscribers with matching filter criteria.

- Guaranteed delivery
  In general, the Notification Service guarantees the delivery of alarm and security alarm notifications to subscribers. An implementation may provide lower quality of service for object creation / deletion, attribute value change and state change notifications. The following error scenarios try to specify the guaranteed delivery in more detail.
  - If the producer fails while it (or one thread of it) is calling *saNtfNotificationSend* then the notification is forwarded either to all subscribers or to no subscriber. Note that it is not intended to block the call of *saNtfNotificationSend* until the notification is forwarded to all subscribers; rather the API function should return as soon as possible after the notification has been passed to the underlying forwarding layer.
  - If an implementation of the Notification Service has one or more instances of separate server processes and the notification library fails to forward a produced notification then the notification library will use temporary storage to avoid that the notification is lost. In error situations like communication outage between library and server or failure of the server either the library of the server or both will make sure that notifications in the temporary storage are forwarded as soon as possible.
  - If an implementation of the Notification Service has one or more instances of separate server processes and one of them fails while it is forwarding a notification to subscribers then the process of forwarding is completed either when this server process has been restarted or failed over to another instance of the server process. Put in other words, the notification will be forwarded to all subscribers even though a server process fails in the middle of forwarding the notification.
  - If an implementation of the Notification Service has one or more instances of separate server processes and there is a notification generated by a producer while one of the server processes has failed, the notification will be forwarded to all subscribers when the server process has been restarted or failed over to another instance.
  - If a notification cannot be forwarded to the logging service then the instance that does the forwarding to the logging service (depending on the implementation this could be either the notification library or a notification server process) will use temporary storage to avoid that the notification is lost and will retry forwarding the notification to the logging service.
    Note that an implementation that does not have a notification server process

has to provide the retry functionality inside the library. If an application is a producer, but not a subscriber at the same time, then it need not call *saNtfDispatch*. Under these conditions retry attempts might occur only when the application calls *saNtfNotificationSend* the next time. This might lead to substantial delay in logging the notification.

- If a subscriber is too slow in reading the notifications that were forwarded to it, the information about the notifications that could not be delivered are forwarded to it by the *SaNtfNotificationDiscardedCallbackT* callback. For alarm notifications and security alarm notifications the list of notification identifiers is provided. The subscriber can use the Reader API to retrieve these notifications by their notification identifier. For other notification types only the amount of notifications that could not be delivered is provided.
  For dropped notifications for which the Notification Service provides the notification identifiers (alarm notifications and security alarm notifications) it is important that the *SaNtfNotificationDiscardedCallbackT* callback is called in the correct chronological order with respect to the regular notification callback (i.e., *SaNtfNotificationCallbackT*). This allows the subscriber to get all of these notifications (i.e., the delivered ones and the dropped ones) in the correct chronological order. For the other notification types, an implementation of the Notification Service may choose to provide the amount of dropped notifications by calling the *SaNtfNotificationDiscardedCallbackT* callback at any time.
- If an implementation of the Notification Service has one or more instances of separate server processes and one of them is temporarily too slow in forwarding notifications to subscribers or the communication channel that is used internally by a Notification Service implementation (and which is currently not specified) is temporarily not available or congested, the service implementation must use mechanisms (like re-transmission of notifications) to avoid lost notifications.
- If a subscriber fails, its subscription for notifications is automatically canceled. If not all forwarded notifications have been delivered to notification callbacks (i.e., *SaNtfNotificationCallbackT*) these remaining notifications are implicitly discarded by the Notification Service. It is the responsibility of the subscriber to checkpoint on the delivered notifications. When it is restarted after failure or failed over to another instance it can use the Reader API to retrieve alarm notifications or security alarm notifications that have occurred after the subscriber's latest checkpoint.
  The automatic subscription cancellation also implies that no new notifications can be forwarded to the failed subscriber before it restarts and subscribes again.
- If a cluster node fails while a notification is being forwarded, its delivery to the subscribers is not guaranteed.

- At most once delivery
  The Notification Service must not deliver a notification to the same subscriber multiple times.
- Ordering
  For a given notification type the notifications are received by subscribers in the same order they were generated by the producer. Likewise, a user of the Reader API when reading logged notifications in chronological order retrieves the notifications of a given notification type in the same order as they were generated by the producer. Since an implementation could use separate communication channels for the different notification types the same order across different notification types cannot be guaranteed.
  Note that an implementation need not guarantee that notifications generated by ***multiple*** producers will always be forwarded to subscribers or logged in the exact chronological order in which they were generated. In a distributed implementation, when more than one producer generates notifications at the same time it is not predictable in which order they will arrive at the subscriber. Under certain conditions, e.g., due to extremely different load levels of the communication layer on different cluster nodes it might happen that a notification generated at time *t* + *x* arrives earlier at a subscriber than another notification created at time *t*, but by a different producer on another node with currently extremely high load. The same is true also for discarded notifications, i.e., the invocation of *SaNtfNotificationDiscardedCallbackT* callback need not be in the exact chronological order in which the notifications were generated.
- Completeness
  Only complete notifications are delivered to a subscriber or a reader. For example, if the producer crashes while it (or one thread of it) is calling *saNtfNotificationSend* then either the complete notification or no notification is forwarded to the subscribers.
- Persistence
  Alarm notifications and security alarm notifications must be stored persistently (while object creation / deletion, attribute value change and state change notifications need not be stored persistently). The Reader API allows to retrieve the logged notifications. This is particularly important for some of the above described error scenarios where a subscriber needs to recover missed notifications.

3.6.1 Discarded Notifications

Normally all notifications matching the filter criteria specified at subscription time are forwarded to a subscriber. For the following reasons related to abnormal behavior of a subscriber or specific runtime conditions notifications are discarded:

(1) The subscriber is too slow in reading notifications via *SaNtfNotificationCallbackT*.

(2) The subscriber process fails (crashes).

(3) The subscriber process unsubscribes without having processed all notifications that were already forwarded to it.

Among the above cases (1) is the only situation where it makes sense to inform the subscriber about discarded notifications. This is done by the *SaNtfNotificationDiscardedCallbackT* callback that may be specified by the subscriber. When this callback is invoked the subscriber may recover either the notifications by using the Reader API (in case of discarded alarm or security alarm notifications) or by retrieving object information (in case of discarded notifications of another notification type).

In case (2) the subscriber does no longer exist. After restart or failover to another process, the new subscriber process can synchronize with the list of notifications using the Reader API or by retrieving object information.

In case (3), when the subscriber is no longer interested in receiving notifications, discarding those notifications that have not been processed by the subscriber while it unsubscribes will do no harm. Otherwise, if the intention actually is to change filter criteria of a subscription, then the subscriber should first subscribe with the new filter criteria and then unsubscribe from the previous subscription (with the old filter criteria).

3.7 Integration of HPI Events

Integration of HPI events is achieved by a dedicated “listener” that acts as an HPI user and receives HPI events via the HPI API *saHpiEventGet()* and converts the event contents into a format appropriate for the Producer API of the Notification Service, i.e., the contents are transferred from the HPI event structure to a notification structure.

The specification of the HPI integration is not within the scope of this document.

3.8 Semantic Identification of Notification Elements

A subset of the above notification parameters are generic containers for elements of varying data type and meaning. As an example the additional information parameter of one notification instance may contain a string representing a file name, while the additional information parameter of another notification instance may contain a string representing a user name. Thus, not only the data type – in this case ‘string’ – but also the meaning of the parameter element has to be specified in the additional information parameter in order to enable subscribers to interpret this element correctly. Such a semantic identifier is needed for the following notification parameters (elements):

- Additional Information element (all notification types)
- Specific Problems element (alarm notifications)
- Threshold Information (alarm notifications)
- Proposed Repair Actions element (alarm notifications)
- Monitored Attributes element (alarm notifications)
- Attribute List element (object create/delete notifications)
- Changed Attribute List element (attribute value change notifications)
- Changed State Attribute List element (state change notifications)

The semantic identifier is called Notification Element Identifier (NEI) from now on, and it is defined to be specific for a notification class and a parameter. Thus, a simple and small numeric identifier will be sufficient in most cases.

Uniqueness of identifiers for each parameter in a notification class is a minimum requirement; a user of the producer API may apply a more restrictive numbering scheme, for instance, with a global numbering scheme where identifiers are unique over all parameters in all notification classes.

The specific problems elements (see *SaNtfSpecificProblemT*) need a special handling concerning the Notification Element Identifier.

3.9 Internationalization Issues

The structure of notifications is suitable for analysis by automated computer-based tools, but it is ill suited for interpretation by human beings. A human reader prefers a concise textual description of the situation in the human language of his choice. In order to support simultaneous use of different languages by different users, localization to the specific language cannot be carried out directly in the notification producer API but must be delayed until the chosen language of the human user is known.

Presenting notification contents at a human interface can certainly be achieved in a generic way, where fixed textual templates are used for each event type, for instance, “New object created” for object creation notifications. A more user-friendly interface uses specific texts for each kind of situation that is shown. This is here achieved by using the notification class identifier as a starting point, defining a specific text for each NCI. In order to get a concise textual description of the situation, each specific text may then reference those notification parameters that are most important for describing that situation. The detailed syntax can be found in 4.2 on page 119.

Note that the internationalization mechanism provided by the Notification Service does not translate any of the notification parameter values (e.g., the additional text parameter or other character string parameters). Rather it provides a link between a

NCI and localized text related to that NCI. However, the localized text can contain variable parts, which are references to notification parameter values.

3.10 API Design Goals

The following design goals were followed when the API of the Notification Service was specified:

- ITU-T X.7xx recommendations
  Most of the attributes specified by the related ITU-T recommendations X.730, X.731, X.733 and X.736 are part of the C structures in this API. The guideline was to “follow in spirit, not in word”. Some attributes were added, such as notifying object or the functionality of internationalization.
- Easy handling of array parameters with variable length
  The data structures of the Notification Service API are quite complex, which is a consequence of the relationship with the ITU-T recommendations. In particular, there are several attributes that are in fact arrays of variable length, for some of them each array element even is a generic data container. Data structures like these are not at all easy to handle in a C program. Therefore, a set of allocation and free functions exists for notifications and notification filters making the programmer’s life easier.
- PDU-Readiness
  Having complex, hierarchical and nested C structures at the API level is one part of reality. The other one is that an implementation has to transport the data efficiently between communication partners. Typically this transport is done using message buffers or PDUs (processing data units), i.e., actually an array of bytes. Making conversion from nested structures to a byte stream easier was yet another reason for providing allocation and free functions. They allow an underlying implementation to let the application program directly operate (read, write) on the internally allocated PDUs.

3.11 Include File and Library Name

The following statement containing declarations of data types and function prototypes must be included in the source of an application using the Notification Service API:

#include <saNtf.h>

To use the Notification Service API, an application must be bound with the Notification Service library. On Unix/Linux systems it is recommended to use the following library:

libSaNtf.so

3.12 Type Definitions

3.12.1 Handles

3.12.1.1 SaNtfHandleT

typedef SaUint64T SaNtfHandleT;

The type of the handle supplied by the Notification Service to a process during initialization of the Notification Service library and used by a process when it invokes functions of the Notification Service API so that the Notification Service can recognize the process.

3.12.1.2 SaNtfNotificationHandleT

typedef SaUint64T SaNtfNotificationHandleT;

The type of a handle to the internal notification structure that is used in API calls.

3.12.1.3 SaNtfNotificationFilterHandleT

typedef SaUint64T SaNtfNotificationFilterHandleT;

The type of a handle to the internal notification filter structure that is used in API calls.

3.12.1.4 SaNtfReadHandleT

typedef SaUint64T SaNtfReadHandleT;

The type of a handle that is used in the Reader API.

3.12.2 Callbacks

3.12.2.1 SaNtfCallbacksT

typedef struct {

SaNtfNotificationCallbackT
saNtfNotificationCallback;

SaNtfNotificationDiscardedCallbackT
saNtfNotificationDiscardedCallback;

} SaNtfCallbacksT;

The type of the callbacks structure supplied by a process to the Notification Service that contains the callback functions that the Notification Service can invoke.

3.12.3 SaNtfNotificationTypeT

typedef enum {

SA_NTF_TYPE_OBJECT_CREATE_DELETE = 0x1000,
SA_NTF_TYPE_ATTRIBUTE_CHANGE = 0x2000,
SA_NTF_TYPE_STATE_CHANGE = 0x3000,
SA_NTF_TYPE_ALARM = 0x4000,
SA_NTF_TYPE_SECURITY_ALARM = 0x5000

} SaNtfNotificationTypeT;

This is the enumeration of all notification types.

3.12.4 SaNtfEventTypeT

#define SA_NTF_NOTIFICATIONS_TYPE_MASK 0xF000

This mask can be used to determine the notification type of an event type easily by binary ANDing the event type with SA_NTF_NOTIFICATIONS_TYPE_MASK.

/ Event types enum, these are only generic **
* * types as defined by the X.73x standards */*
typedef enum {

SA_NTF_OBJECT_NOTIFICATIONS_START =
SA_NTF_TYPE_OBJECT_CREATE_DELETE,
SA_NTF_OBJECT_CREATION,
SA_NTF_OBJECT_DELETION,

SA_NTF_ATTRIBUTE_NOTIFICATIONS_START =
SA_NTF_TYPE_ATTRIBUTE_CHANGE,
SA_NTF_ATTRIBUTE_ADDED,
SA_NTF_ATTRIBUTE_REMOVED,
SA_NTF_ATTRIBUTE_CHANGED,
SA_NTF_ATTRIBUTE_RESET,

SA_NTF_STATE_CHANGE_NOTIFICATIONS_START =
SA_NTF_TYPE_STATE_CHANGE,
SA_NTF_OBJECT_STATE_CHANGE,

SA_NTF_ALARM_NOTIFICATIONS_START = SA_NTF_TYPE_ALARM,
SA_NTF_ALARM_COMMUNICATION,
SA_NTF_ALARM_QOS,
SA_NTF_ALARM_PROCESSING,

```
        SA_NTF_ALARM_EQUIPMENT,
        SA_NTF_ALARM_ENVIRONMENT,

        SA_NTF_SECURITY_ALARM_NOTIFICATIONS_START =
              SA_NTF_TYPE_SECURITY_ALARM,
        SA_NTF_INTEGRITY_VIOLATION,
        SA_NTF_OPERATION_VIOLATION,
        SA_NTF_PHYSICAL_VIOLATION,
        SA_NTF_SECURITY_SERVICE_VIOLATION,
        SA_NTF_TIME_VIOLATION
} SaNtfEventTypeT;
```

SaNtfEventTypeT defines all event types that are allowed in notifications.

3.12.5 Notification Object

Use *SaNameT*. This will typically be LDAP DNs defined by AIS, e.g., for an AMF component or a message queue of the Message Service or other AIS objects. Currently, the Notification Service does not define a naming scheme for non-AIS objects such as resources of the operating system, HPI objects or application-specific objects. The value of the notification object is interpreted by the Notification Service only for those cases defined in *Filtering* on page 85.

3.12.6 Notifying Object

Use *SaNameT*. This will typically be the LDAP DN of an AMF logical entity producing the notification. If the notifying object is not an AMF logical entity an application specific notation may be used instead. Currently, the Notification Service does not define a naming scheme for notifying objects, which are not AMF components. The value of the notification object is interpreted by the Notification Service only for those cases defined in *Filtering* on page 85.

3.12.7 SaNtfClassIdT

typedef struct {

SaUint32T vendorId;
SaUint16T majorId;
SaUint16T minorId;

} SaNtfClassIdT;

This is the notification class identifier, which uniquely identifies the kind of situation that caused the notification. This identifier alone is sufficient to identify the kind of situation, no other information from the notification is necessary. For *vendorId* it is suggested to use the SNMP enterprise number as listed in [7]. The *majorId* and *minorId* values can be arbitrarily assigned to a NCI by a vendor.

#define SA_NTF_VENDOR_ID_SAF 18568

This is a predefined *vendorId* for those NCIs specified by SA Forum. The SNMP enterprise number of SA Forum is taken here. See *SaServicesT* in the SA Forum Overview document [5] for the pre-defined values of *majorId* of the SA Forum services.

3.12.8 SaServicesT

Defined in the SA Forum Overview document [5]. This enumeration defines the values for the SA Forum services as used for *majorId* in *SaNtfClassIdT*, i.e., the values used for *majorId* when *vendorId* is *SA_NTF_VENDOR_ID_SAF*.

3.12.9 SaNtfElementIdT

typedef SaUint16T SaNtfElementIdT;

This is the data type of the Notification Element Identifier (NEI). A value is scoped to a Notification Class Identifier (NCI).

3.12.10 SaNtfIdentifierT

typedef SaUint64T SaNtfIdentifierT;

This type is used for notification identifiers.

#define SA_NTF_IDENTIFIER_UNUSED ((SaNtfIdentifierT) 0)

The special value of *SA_NTF_IDENTIFIER_UNUSED* has to be used to indicate that a variable of the type *SaNtfIdentifierT* does not contain a valid notification identifier.

3.12.11 Event Time

Use *SaTimeT*.

3.12.12 SaNtfValueTypeT

```
typedef enum {
    SA_NTF_VALUE_UINT8,      /* A byte long - unsigned int */
    SA_NTF_VALUE_INT8,       /* A byte long - signed int */
    SA_NTF_VALUE_UINT16,     /* 2 bytes long - unsigned int */
    SA_NTF_VALUE_INT16,      /* 2 bytes long - signed int */
    SA_NTF_VALUE_UINT32,     /* 4 bytes long - unsigned int */
    SA_NTF_VALUE_INT32,      /* 4 bytes long - signed int */
    SA_NTF_VALUE_FLOAT,      /* 4 bytes long - float */
    SA_NTF_VALUE_UINT64,     /* 8 bytes long - unsigned int */
    SA_NTF_VALUE_INT64,      /* 8 bytes long - signed int */
    SA_NTF_VALUE_DOUBLE,     /* 8 bytes long - double */
    SA_NTF_VALUE_LDAP_NAME,  /* SaNameT type */
    SA_NTF_VALUE_STRING,     /* '\0' terminated char array (UTF-8
                                encoded) */
    SA_NTF_VALUE_IPADDRESS,  /* IPv4 or IPv6 address as '\0' terminated
                                char array */
    SA_NTF_VALUE_BINARY,     /* Binary data stored in bytes - number of
                                bytes stored separately */
    SA_NTF_VALUE_ARRAY,      /* Array of some data type - size of elements
                                and number of elements stored separately */
} SaNtfValueTypeT;
```

SaNtfValueTypeT defines the possible types of those values within a structure of type *SaNtfValueT*.

3.12.13 SaNtfValueT

```
typedef union {
    /* The first few are fixed size data types*/
    SaUint8T    uint8Val;    /* SA_NTF_VALUE_UINT8 */
    SaInt8T     int8Val;     /* SA_NTF_VALUE_INT8 */
    SaUint16T   uint16Val;   /* SA_NTF_VALUE_UINT16 */
    SaInt16T    int16Val;    /* SA_NTF_VALUE_INT16 */
    SaUint32T   uint32Val;   /* SA_NTF_VALUE_UINT32 */
    SaInt32T    int32Val;    /* SA_NTF_VALUE_INT32 */
    SaFloatT    floatVal;    /* SA_NTF_VALUE_FLOAT */
    SaUint64T   uint64Val;   /* SA_NTF_VALUE_UINT64 */
    SaInt64T    int64Val;    /* SA_NTF_VALUE_INT64 */
    SaDoubleT   doubleVal;   /* SA_NTF_VALUE_DOUBLE */

    /* This struct can represent variable length fields like*
    * LDAP names, strings, IP addresses and binary data. *
    * It may only be used in conjunction with the data type values *
    * SA_NTF_VALUE_LDAP_NAME, SA_NTF_VALUE_STRING, *
    * SA_NTF_VALUE_IPADDRESS and SA_NTF_VALUE_BINARY.
    * This field shall not be directly accessed. *
    * To initialize this structure and to set a pointer to the real data*
    * use saNtfPtrValAllocate(). The function saNtfPtrValGet() shall be used *
    * for retrieval of the real data.
    */
    struct {
        SaUint16T dataOffset;
        SaUint16T dataSize;
    } ptrVal;

    /* This struct represents sets of data of identical type*
    * like notification identifiers, attributes etc. *
    * It may only be used in conjunction with the data type value *
    * SA_NTF_VALUE_ARRAY. Functions *
    * SaNtfArrayValAllocate() or SaNtfArrayValGet() shall be used to*
    * get a pointer for accessing the real data. Direct access is not allowed. */
    struct {
        SaUint16T arrayOffset;
        SaUint16T numElements;
```

```
        SaUint16T elementSize;
    } arrayVal;
} SaNtfValueT;
```

SaNtfValueT defines a structure that is used in notifications for parameters or parameter elements that may be of varying data type. A value could be one of the types specified by *SaNtfValueTypeT*.

SaNtfValueT defines fields for several simple data types, like *SA_NTF_VALUE_INT16* or *SA_NTF_VALUE_DOUBLE*. These simple data types can be stored directly in the *SaNtfValueT* union. However, for other data types, e.g., *SA_NTF_VALUE_STRING* or *SA_NTF_VALUE_ARRAY*, *SaNtfValueT* cannot hold the memory needed to store the actual data; for those data types rather additional memory *outside* from *SaNtfValueT* has to be reserved. This is done by either *saNtfPtrValAllocate* or *saNtfArrayValAllocate.* These allocation functions use the *ptrVal* or *arrayVal* field in *SaNtfValueT* respectively to store reference and size information related to the reserved memory.

An application may not interpret the contents of the *ptrVal* or *arrayVal* fields in *SaNtfValueT* in order to access the memory directly. Rather an application is supposed to only access memory via the data pointers returned from the allocation functions (*saNtfPtrValAllocate* or *saNtfArrayValAllocate*) or the related get functions (*saNtfPtrValGet* or *saNtfArrayValGet*).

3.12.14 Additional Text

Use *SaStringT*. A string consists of UTF-8 encoded characters and is terminated by the ‘\0’ character.

3.12.15 SaNtfAdditionalInfoT

```
typedef struct {
    SaNtfElementIdT infoId;
    /* API user is expected to define this field*/
    SaNtfValueTypeT infoType;
    SaNtfValueT infoValue;
} SaNtfAdditionalInfoT;
```

This structure represents a single element in the additional information parameter of a notification.

3.12.16 SaNtfProbableCauseT

This is the enumeration of probable causes as described in X.733 [3] and X.736 [4].

typedef enum {

SA_NTF_ADAPTER_ERROR,
SA_NTF_APPLICATION_SUBSYSTEM_FAILURE,
SA_NTF_BANDWIDTH_REDUCED,
SA_NTF_CALL_ESTABLISHMENT_ERROR,
SA_NTF_COMMUNICATIONS_PROTOCOL_ERROR,
SA_NTF_COMMUNICATIONS_SUBSYSTEM_FAILURE,
SA_NTF_CONFIGURATION_OR_CUSTOMIZATION_ERROR,
SA_NTF_CONGESTION,
SA_NTF_CORRUPT_DATA,
SA_NTF_CPU_CYCLES_LIMIT_EXCEEDED,
SA_NTF_DATASET_OR_MODEM_ERROR,
SA_NTF_DEGRADED_SIGNAL,
SA_NTF_D_T_E,
SA_NTF_ENCLOSURE_DOOR_OPEN,
SA_NTF_EQUIPMENT_MALFUNCTION,
SA_NTF_EXCESSIVE_VIBRATION,
SA_NTF_FILE_ERROR,
SA_NTF_FIRE_DETECTED,
SA_NTF_FLOOD_DETECTED,
SA_NTF_FRAMING_ERROR,
*SA_NTF_HEATING_OR_VENTILATION_OR_COOLING_*
SYSTEM_PROBLEM,
SA_NTF_HUMIDITY_UNACCEPTABLE,
SA_NTF_INPUT_OUTPUT_DEVICE_ERROR,
SA_NTF_INPUT_DEVICE_ERROR,
SA_NTF_L_A_N_ERROR,
SA_NTF_LEAK_DETECTED,
SA_NTF_LOCAL_NODE_TRANSMISSION_ERROR,
SA_NTF_LOSS_OF_FRAME,
SA_NTF_LOSS_OF_SIGNAL,
SA_NTF_MATERIAL_SUPPLY_EXHAUSTED,
SA_NTF_MULTIPLEXER_PROBLEM,
SA_NTF_OUT_OF_MEMORY,
SA_NTF_OUTPUT_DEVICE_ERROR,
SA_NTF_PERFORMANCE_DEGRADED,
SA_NTF_POWER_PROBLEM,
SA_NTF_PRESSURE_UNACCEPTABLE,
SA_NTF_PROCESSOR_PROBLEM,

```
        SA_NTF_PUMP_FAILURE,
        SA_NTF_QUEUE_SIZE_EXCEEDED,
        SA_NTF_RECEIVE_FAILURE,
        SA_NTF_RECEIVER_FAILURE,
        SA_NTF_REMOTE_NODE_TRANSMISSION_ERROR,
        SA_NTF_RESOURCE_AT_OR_NEARING_CAPACITY,
        SA_NTF_RESPONSE_TIME_EXCESSIVE,
        SA_NTF_RETRANSMISSION_RATE_EXCESSIVE,
        SA_NTF_SOFWARE_ERROR,
        SA_NTF_SOFWARE_PROGRAM_ABNORMALLY_TERMINATED,
        SA_NTF_SOFTWARE_PROGRAM_ERROR,
        SA_NTF_STORAGE_CAPACITY_PROBLEM,
        SA_NTF_TEMPERATURE_UNACCEPTABLE,
        SA_NTF_THRESHOLD_CROSSED,
        SA_NTF_TIMING_PROBLEM,
        SA_NTF_TOXIC_LEAK_DETECTED,
        SA_NTF_TRANSMIT_FAILURE,
        SA_NTF_TRANSMITTER_FAILURE,
        SA_NTF_UNDERLYING_RESOURCE_UNAVAILABLE,
        SA_NTF_VERSION_MISMATCH,

        SA_NTF_AUTHENTICATION_FAILURE,
        SA_NTF_BREACH_OF_CONFIDENTIALITY,
        SA_NTF_CABLE_TAMPER,
        SA_NTF_DELAYED_INFORMATION,
        SA_NTF_DENIAL_OF_SERVICE,
        SA_NTF_DUPLICATE_INFORMATION,
        SA_NTF_INFORMATION_MISSING,
        SA_NTF_INFORMATION_MODIFICATION_DETECTED,
        SA_NTF_INFORMATION_OUT_OF_SEQUENCE,
        SA_NTF_INTRUSION_DETECTION,
        SA_NTF_KEY_EXPIRED,
        SA_NTF_NON_REPUDIATION_FAILURE,
        SA_NTF_OUT_OF_HOURS_ACTIVITY,
        SA_NTF_OUT_OF_SERVICE,
        SA_NTF_PROCEDURAL_ERROR,
        SA_NTF_UNAUTHORIZED_ACCESS_ATTEMPT,
        SA_NTF_UNEXPECTED_INFORMATION,
        SA_NTF_UNSPECIFIED_REASON
} SaNtfProbableCauseT;
```

3.12.17 SaNtfSpecificProblemT

typedef struct {

SaNtfElementIdT problemId;
/ API user is expected to define this field*/*

SaNtfClassIdT problemClassId;
/ optional field to identify problemId values from other NCIs,*
*needed for correlation between clear and non-clear alarms */*

SaNtfValueTypeT problemType;

SaNtfValueT problemValue;

} SaNtfSpecificProblemT;

This structure represents a single element in the specific problem parameter of a notification. The field *problemClassId* is optional. If it is not specified (all fields of *problemClassId* are 0), the *problemId* value is local to the NCI of the notification. If it is specified, the given *problemId* value is taken from the NCI given by *problemClassId*. If an alarm notification of perceived severity SA_NTF_SEVERITY_CLEARED contains a non-empty *specificProblems* parameter, then the field *problemClassId* of each element in that parameter must be filled in to reference the notification element identifier of the alarm that is to be cleared.

3.12.18 SaNtfSeverityT

This is the enumeration for severities used by alarm notifications and security alarm notifications. Security alarm notifications use a subset of the values, only.

typedef enum {

SA_NTF_SEVERITY_CLEARED, / alarm notification, only */*
SA_NTF_SEVERITY_INDETERMINATE,
SA_NTF_SEVERITY_WARNING,
SA_NTF_SEVERITY_MINOR,
SA_NTF_SEVERITY_MAJOR,
SA_NTF_SEVERITY_CRITICAL

} SaNtfSeverityT;

3.12.19 SaNtfSeverityTrendT

This is the enumeration for trend indication of severity.

```
typedef enum {
        SA_NTF_TREND_MORE_SEVERE,
        SA_NTF_TREND_NO_CHANGE,
        SA_NTF_TREND_LESS_SEVERE
} SaNtfSeverityTrendT;
```

3.12.20 SaNtfThresholdInformationT

```
typedef struct {
        SaNtfElementIdT thresholdId;
        /* API user is expected to define this field*/
        SaNtfValueTypeT thresholdValueType;
        SaNtfValueT thresholdValue;
        SaNtfValueT thresholdHysteresis;
        /* This has to be of the same type as threshold */
        SaNtfValueT observedValue;
        SaTimeT armTime;
} SaNtfThresholdInformationT;
```

This structure contains information about a triggered threshold. The *thresholdValue*, *thresholdHystereris* and the *observedValue* have to be of the same data type, defined by the field *thresholdValueType*.

3.12.21 SaNtfProposedRepairActionT

typedef struct {

SaNtfElementIdT actionId;
/ API user is expected to define this field*/*

SaNtfValueTypeT actionValueType;

SaNtfValueT actionValue;

} SaNtfProposedRepairActionT;

Structure to represent a single proposed repair action in an alarm notification.

Currently, SA Forum does not specify any mechanism to define an association between a single proposed repair action and a single specific problem.

3.12.22 SaNtfSourceIndicatorT

typedef enum {

SA_NTF_OBJECT_OPERATION = 1,
SA_NTF_MANAGEMENT_OPERATION = 2,
SA_NTF_UNKNOWN_OPERATION = 3

} SaNtfSourceIndicatorT;

This is the source indicator for state change, object create/delete and attribute value change notifications.

3.12.23 SaNtfStateChangeT

typedef struct {

SaNtfElementIdT stateId;

SaBoolT oldStatePresent;
SaUint16T oldState;

SaUint16T newState;

} SaNtfStateChangeT;

Structure to represent state changes as part of a notification. The *oldState* and *newState* fields contain the old and new state value and the stateId field identifies the kind of state that has changed. The values of stateId are defined in the scope of a NCI. The value of the optional field *oldState* is relevant only when *oldStatePresent* is SA_TRUE.

3.12.24 SaNtfAttributeT

```
typedef struct {
    SaNtfElementIdT attributeId;
    /* API user is expected to define this field*/
    SaNtfValueTypeT attributeType;
    SaNtfValueT attributeValue;
} SaNtfAttributeT;
```

This is the structure to represent object attributes in an object creation or deletion notification.

3.12.25 SaNtfAttributeChangeT

```
typedef struct {
    SaNtfElementIdT attributeId;
    /* API user is expected to define this field*/
    SaNtfValueTypeT attributeType;
    SaBoolT oldAttributePresent;
    SaNtfValueT oldAttributeValue;
    SaNtfValueT newAttributeValue;
} SaNtfAttributeChangeT;
```

This is the structure to represent attribute changes in a notification. The values of attributeId are defined in the scope of a NCI. The value of the optional field *oldAttributeValue* is relevant only when *oldAttributePresent* is SA_TRUE.

3.12.26 SaNtfServiceUserT

```
typedef struct {
    SaNtfValueTypeT valueType;
    SaNtfValueT value;
} SaNtfServiceUserT;
```

This is the structure to represent the service user and service provider in a security alarm notification.

3.12.27 SaNtfSecurityAlarmDetectorT

typedef struct {

SaNtfValueTypeT valueType;

SaNtfValueT value;

} SaNtfSecurityAlarmDetectorT;

This is the structure to represent the security alarm detector in a security alarm notification.

3.12.28 SaNtfNotificationHeaderT

This structure has pointers pointing to the common fields in the internal notification structure.

typedef struct {

*SaNtfEventTypeT *eventType;*
/ This points to the event type in**
*** the internal notification structure*/*

*SaNameT *notificationObject;*
/ This points to the notification object**
*** in the internal notification structure*/*

*SaNameT *notifyingObject;*
/ This points to the notifying object *
*** in the internal notification structure */*

*SaNtfClassIdT *notificationClassId;*
/ This points to the notification class identifier */*

*SaTimeT *eventTime;*
/ Points to eventTime*/*

SaUint16T numCorrelatedNotifications;
/ Number of correlated notifications in the notification*/*

SaUint16T lengthAdditionalText;
/ Length of additional text in bytes (including terminating '\0')*/*

SaUint16T numAdditionalInfo;
/ Number of additional info fields*/*

*SaNtfIdentifierT *notificationId;*
/ Points to the notification ID in**
*** the internal notification structure*/*

```
    SaNtfIdentifierT *correlatedNotifications;
    /* Points to the correlated*
     * notification identifiers array*/

    SaStringT additionalText;
    /* Points to the additional text in*
     * the internal notification structure(\0 terminated, UTF-8 encoded) */

    SaNtfAdditionalInfoT *additionalInfo;
    /* Points to the additional info array in*
     * the internal notification structure*/
} SaNtfNotificationHeaderT;
```

3.12.29 SaNtfObjectCreateDeleteNotificationT

This structure contains pointers to the fields in an object create/delete notification.

```
typedef struct {
    SaNtfNotificationHandleT notificationHandle;
    * A handle to the internal notification structure*/

    SaNtfNotificationHeaderT notificationHeader;
    /* Notification header*/

    SaUint16T numAttributes;
    /* Number of object attributes in the notification*/

    SaNtfSourceIndicatorT *sourceIndicator;
    /* Points to the source indicator*
     * field in the internal notification structure*/

    SaNtfAttributeT *objectAttributes;
    /* Pointer to attributes array in the internal notification structure*/
} SaNtfObjectCreateDeleteNotificationT;
```

3.12.30 SaNtfAttributeChangeNotificationT

This structure contains pointers to the fields in an attribute change notification.

```
typedef struct {
    SaNtfNotificationHandleT notificationHandle;
    /* A handle to the internal notification structure*/
    SaNtfNotificationHeaderT notificationHeader;
    /* Notification header*/
    SaUint16T numAttributes;
    /* Number of changed attributes in the notification*/
    SaNtfSourceIndicatorT *sourceIndicator;
    /* Points to the source indicator*
     * field in the internal notification structure*/
    SaNtfAttributeChangeT *changedAttributes;
    /* Points to changed attributes*
     * array in the internal notification structure*/
} SaNtfAttributeChangeNotificationT;
```

3.12.31 SaNtfStateChangeNotificationT

This structure has pointers to the fields in a state change notification.

```
typedef struct {
    SaNtfNotificationHandleT notificationHandle;
    /* A handle to the internal notification structure*/
    SaNtfNotificationHeaderT notificationHeader;
    /* Notification header*/
    SaUint16T numStateChanges;
    /* Number of state changes in the notification*/
    SaNtfSourceIndicatorT *sourceIndicator;
    /* Points to the source indicator*
     * field in the internal notification structure*/
    SaNtfStateChangeT *changedStates;
    /* Points to changed states array in the internal notification structure*/
} SaNtfStateChangeNotificationT;
```

3.12.32 SaNtfAlarmNotificationT

This structure contains pointers to the fields in an alarm notification.

```
typedef struct {
    SaNtfNotificationHandleT notificationHandle;
    /* A handle to the internal notification structure */
    SaNtfNotificationHeaderT notificationHeader;
    /* Notification header*/
    SaUint16T numSpecificProblems;
    /* Number of specific problems*/
    SaUint16T numMonitoredAttributes;
    /* Number of monitored attributes*/
    SaUint16T numProposedRepairActions;
    /* Number of proposed repair actions*/
    SaNtfProbableCauseT *probableCause;
    /* Points to the probable cause field*/
    SaNtfSpecificProblemT *specificProblems;
    /* Points to the array of specific problems*/
    SaNtfSeverityT *perceivedSeverity;
    /* Points to perceived severity*/
    SaNtfSeverityTrendT *trend;
    /* Points to trend of severity*/
    SaNtfThresholdInformationT *thresholdInformation;
    /* Points to the threshold information field*/
    SaNtfAttributeT *monitoredAttributes;
    /* Monitored attributes array*/
    SaNtfProposedRepairActionT *proposedRepairActions;
    / * Proposed repair actions array */
} SaNtfAlarmNotificationT;
```

3.12.33 SaNtfSecurityAlarmNotificationT

This structure contains pointers to the fields in security alarm notification.

```
typedef struct {
    SaNtfNotificationHandleT notificationHandle;
    /* A handle to the internal notification structure */
    SaNtfNotificationHeaderT notificationHeader;
    /* Notification header*/
    SaNtfProbableCauseT *probableCause;
    /* Points to the probable cause field*/
    SaNtfSeverityT *severity;
    /* Points to severity field*/
    SaNtfSecurityAlarmDetectorT *securityAlarmDetector;
    /* Pointer to the alarm detector field*/
    SaNtfServiceUserT*serviceUser;
    /* Pointer to the service user field*/
    SaNtfServiceUserT *serviceProvider;
    /* Pointer to the service user field*/
} SaNtfSecurityAlarmNotificationT;
```

3.12.34 Default variable notification data size

/* Default for API user when not sure how much memory is in total needed to accommodate the variable size data */

```
#define SA_NTF_ALLOC_SYSTEM_LIMIT(-1)
```

Default value for maximum number of bytes to be specified for accommodating variable size notification data. It can be used when not sure how much memory is in total needed to accommodate the variable size data.

3.12.35 SaNtfSubscriptionIdT

```
typedef SaUint32T SaNtfSubscriptionIdT;
```

The type of an identifier for a particular subscription for notifications by a particular process with a particular notification filter. This identifier is used to associate delivery of notifications for that subscription to the process.

3.12.36 SaNtfNotificationFilterHeaderT

This structure contains filter elements common to all notification types.

```
typedef struct {
    SaUint16T numEventTypes;
    /* number of event types */
    SaNtfEventTypeT *eventTypes;
    /* the array of event types */
    SaUint16T numNotificationObjects;
    /* number of notification objects */
    SaNameT *notificationObjects;
    /* the array of notification objects */
    SaUint16T numNotifyingObjects;
    /* number of notifying objects */
    SaNameT *notifyingObjects;
    /* the array of notifying objects */
    SaUint16T numNotificationClassIds;
    /* number of notification class ids */
    SaNtfClassIdT *notificationClassIds;
    /* the array of notification class ids */
} SaNtfNotificationFilterHeaderT;
```

3.12.37 SaNtfObjectCreateDeleteNotificationFilterT

This structure contains filter elements for an object create/delete notification filter.

```
typedef struct {
    SaNtfNotificationFilterHandleT notificationFilterHandle;
    /* a handle to the internal notification filter structure */
    SaNtfNotificationFilterHeaderT notificationFilterHeader;
     /* the notification filter header */
    SaUint16T numSourceIndicators;
    /* number of source indicators */
    SaNtfSourceIndicatorT *sourceIndicators;
    /* the array of source indicators */
} SaNtfObjectCreateDeleteNotificationFilterT;
```

3.12.38 SaNtfAttributeChangeNotificationFilterT

This structure contains filter elements for an attribute change notification filter.

```
typedef struct {
    SaNtfNotificationFilterHandleT notificationFilterHandle;
    /* a handle to the internal notification filter structure */
    SaNtfNotificationFilterHeaderT notificationFilterHeader;
    /* the notification filter header */
    SaUint16T numSourceIndicators;
    /* number of source indicators */
    SaNtfSourceIndicatorT *sourceIndicators;
    /* the array of source indicators */
} SaNtfAttributeChangeNotificationFilterT;
```

3.12.39 SaNtfStateChangeNotificationFilterT

This structure contains filter elements for a state change notification filter.

```
typedef struct {
    SaNtfNotificationFilterHandleT notificationFilterHandle;
    /* a handle to the internal notification filter structure */
    SaNtfNotificationFilterHeaderT notificationFilterHeader;
    /* the notification filter header */
    SaUint16T numSourceIndicators;
    /* number of source indicators */
    SaNtfSourceIndicatorT *sourceIndicators;
    /* the array of source indicators */
    SaUint16T numStateChanges;
    /* number of state changes */
    SaNtfStateChangeT *changedStates;
    /* the array of changed states */
} SaNtfStateChangeNotificationFilterT;
```

3.12.40 SaNtfAlarmNotificationFilterT

This structure contains filter elements for an alarm notification filter.

```
typedef struct {
    SaNtfNotificationFilterHandleT notificationFilterHandle;
    /* a handle to the internal notification filter structure */
    SaNtfNotificationFilterHeaderT notificationFilterHeader;
    /* the notification filter header */
    SaUint16T numProbableCauses;
    /* number of probable causes */
    SaUint16T numPerceivedSeverities;
    /* number of perceived severities */
    SaUint16T numTrends;
    /* number of severity trends */
    SaNtfProbableCauseT *probableCauses;
    /* the array of probable causes */
    SaNtfSeverityT *perceivedSeverities;
    /* the array of perceived severities */
    SaNtfSeverityTrendT *trends;
    /* the array of severity trends */
} SaNtfAlarmNotificationFilterT;
```

3.12.41 SaNtfSecurityAlarmNotificationFilterT

This structure contains filter elements for a security alarm notification filter.

```
typedef struct {
    SaNtfNotificationFilterHandleT notificationFilterHandle;
    /* a handle to the internal notification filter structure */
    SaNtfNotificationFilterHeaderT notificationFilterHeader;
    /* the notification filter header */
    SaUint16T numProbableCauses;
    /* number of probable causes */
    SaUint16T numSeverities;
    /* number of severities */
    SaUint16T numSecurityAlarmDetectors;
    /* number of security alarm detectors */
    SaUint16T numServiceUsers;
    /* number of service users */
    SaUint16T numServiceProviders;
    /* number of service providers */
    SaNtfProbableCauseT *probableCauses;
    /* the array of probable causes */
    SaNtfSeverityT *severities;
    /* the array of severities */
    SaNtfSecurityAlarmDetectorT *securityAlarmDetectors;
    /* the array of security alarm detectors */
    SaNtfServiceUserT *serviceUsers;
    /* the array of service users */
    SaNtfServiceUserT *serviceProviders;
    /* the array of service providers */
} SaNtfSecurityAlarmNotificationFilterT;
```

3.12.42 SaNtfSearchModeT

```
typedef enum {
    SA_NTF_SEARCH_BEFORE_OR_AT_TIME = 1,
    SA_NTF_SEARCH_AT_TIME = 2,
    SA_NTF_SEARCH_AT_OR_AFTER_TIME = 3,
    SA_NTF_SEARCH_BEFORE_TIME = 4,
    SA_NTF_SEARCH_AFTER_TIME = 5,
    SA_NTF_SEARCH_NOTIFICATION_ID = 6,
    SA_NTF_SEARCH_ONLY_FILTER = 7
} SaNtfSearchModeT;
```

This enumeration defines the search modes for the Reader API.

3.12.43 SaNtfSearchCriteriaT

```
typedef struct {
    SaNtfSearchModeT searchMode;
    /* indicates the search mode */
    SaTimeT eventTime;
    /* event time (relevant only if searchMode is one of
       SA_NTF_SEARCH_*_TIME) */
    SaNtfIdentifierT notificationId;
    /* notification ID (relevant only if searchMode is
       SA_NTF_SEARCH_NOTIFICATION_ID) */
} SaNtfSearchCriteriaT;
```

This structure contains the search criteria for the Reader API.

3.12.44 SaNtfSearchDirectionT

```
typedef enum {
    SA_NTF_SEARCH_OLDER = 1,
    SA_NTF_SEARCH_YOUNGER = 2
} SaNtfSearchDirectionT;
```

This enumeration defines the search directions for the Reader API.

3.12.45 SaNtfNotificationTypeFilterHandlesT

This structure aggregates fields for notification filter handles of all notification types.

```
typedef struct {
    SaNtfNotificationFilterHandleT objectCreateDeleteFilterHandle;
    SaNtfNotificationFilterHandleT attributeChangeFilterHandle;
    SaNtfNotificationFilterHandleT stateChangeFilterHandle;
    SaNtfNotificationFilterHandleT alarmFilterHandle;
    SaNtfNotificationFilterHandleT securityAlarmFilterHandle;
} SaNtfNotificationTypeFilterHandlesT;
```

Unused handles in *SaNtfNotificationTypeFilterHandlesT* shall be set to SA_NTF_FILTER_HANDLE_NULL.

```
#define SA_NTF_FILTER_HANDLE_NULL((SaNtfNotificationFilterHandleT) NULL)
```

3.12.46 SaNtfNotificationsT

```
typedef struct {
    SaNtfNotificationTypeT notificationType;
    union
    {
        SaNtfObjectCreateDeleteNotificationT objectCreateDeleteNotification;
        SaNtfAttributeChangeNotificationT attributeChangeNotification;
        SaNtfStateChangeNotificationT stateChangeNotification;
        SaNtfAlarmNotificationT alarmNotification;
        SaNtfSecurityAlarmNotificationT securityAlarmNotification;
    } notification;
} SaNtfNotificationsT;
```

This structure contains a union of all notification type specific structures.

3.13 Library Life Cycle

3.13.1 saNtfInitialize()

Prototype

SaAisErrorT saNtfInitialize(
*SaNtfHandleT *ntfHandle,*
*const SaNtfCallbacksT *ntfCallbacks,*
*SaVersionT *version*
);

Parameters

ntfHandle - [out] A pointer to the handle designating this particular initialization of the Notification Service that is to be returned by the Notification Service.

ntfCallbacks - [in] If *ntfCallbacks* is set to NULL, no callback is registered; otherwise, it is a pointer to a *SaNtfCallbacksT* structure, containing the callback functions of the process that the Notification Service may invoke. Only non-NULL callback functions in this structure will be registered.

version - [in/out] As an input parameter, *version* is a pointer to the required Notification Service version. In this case, *minorVersion* is ignored and should be set to 0x00. As an output parameter, the version actually supported by the Notification Service is delivered.

Description

This function initializes the Notification Service for the invoking process and registers the various callback functions. This function must be invoked prior to the invocation of any other Notification Service functionality. The handle *ntfHandle* is returned as the reference to this association between the process and the Notification Service. The process uses this handle in subsequent communication with the Notification Service. If the implementation supports the required *releaseCode*, and a major version >= the required *majorVersion*, SA_AIS_OK is returned. In this case, the *version* parameter is set by this function to:

- *releaseCode* = required release code
- *majorVersion* = highest value of the major version that this implementation can-support for the required *releaseCode*
- *minorVersion* = highest value of the minor version that this implementation can support for the required value of *releaseCode* and the returned value of *majorVersion*

If the above mentioned condition cannot be met, SA_AIS_ERR_VERSION is returned, and the version parameter is set to:

if (implementation supports the required releaseCode)

releaseCode = required releaseCode

else {

if (implementation supports releaseCode higher than the required releaseCode)

releaseCode = the least value of the supported release codes that is higher than the required releaseCode

else

releaseCode = the highest value of the supported release codes that is less than the required releaseCode

}

majorVersion = highest value of the major versions that this implementation can support for the returned *releaseCode*

minorVersion = highest value of the minor versions that this implementation can support for the returned values of *releaseCode* and *majorVersion*

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_VERSION - The *version* parameter is not compatible with the version of the Notification Service implementation.

See Also

saNtfSelectionObjectGet(), saNtfDispatch(), saNtfFinalize()

3.13.2 saNtfSelectionObjectGet()

Prototype

```
SaAisErrorT saNtfSelectionObjectGet(
        SaNtfHandleT ntfHandle,
        SaSelectionObjectT *selectionObject
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

selectionObject - [out] A pointer to the operating system handle that the invoking process can use to detect pending callbacks.

Description

This function returns the operating system handle, *selectionObject*, associated with the handle *ntfHandle*. The invoking process can use this handle to detect pending callbacks, instead of repeatedly invoking *saNtfDispatch()* for this purpose.

In a POSIX environment, the operating system handle is a file descriptor that is used with the *poll()* or *select()* system calls to detect incoming callbacks.

The *selectionObject* returned by *saNtfSelectionObjectGet()* is valid until *saNtfFinalize()* is invoked on the same handle *ntfHandle*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle ntfHandle is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfInitialize(), saNtfDispatch(), saNtfFinalize()

3.13.3 saNtfDispatch()

Prototype

SaAisErrorT saNtfDispatch(
SaNtfHandleT ntfHandle,
SaDispatchFlagsT dispatchFlags
);

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

dispatchFlags - [in] Flags that specify the callback execution behavior of the *saNtfDispatch()* function, which have the values SA_DISPATCH_ONE, SA_DISPATCH_ALL, or SA_DISPATCH_BLOCKING, as defined in the SA Forum Overview document.

Description

This function invokes, in the context of the calling thread, pending callbacks for the handle *ntfHandle* in a way that is specified by the *dispatchFlags* parameter.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle ntfHandle is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - The dispatchFlags parameter is invalid.

See Also

saNtfInitialize(), saNtfFinalize()

3.13.4 saNtfFinalize()

Prototype

SaAisErrorT saNtfFinalize(
SaNtfHandleT ntfHandle
);

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

Description

The *saNtfFinalize()* function closes the association, represented by the *ntfHandle* parameter, between the invoking process and the Notification Service. The process must have invoked *saNtfInitialize()* before it invokes this function. A process must invoke this function once for each handle it acquired by invoking *saNtfInitialize()*.

If the *saNtfFinalize()* function returns successfully, the *saNtfFinalize()* function releases all resources acquired when *saNtfInitialize()* was called. Moreover, it closes all notification channels that are open for the particular handle. Furthermore, it cancels all pending callbacks related to the particular handle. Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

After *saNtfFinalize()* is called, the selection object is no longer valid.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle ntfHandle is invalid, since it is corrupted, uninitialized, or has already been finalized.

See Also

saNtfInitialize()

3.14 Operations of the Producer API

This section describes the API functions that enable the caller to generate notifications. Generation of notifications is divided into four steps:

1. Allocating memory for the notification contents with one of the allocation functions described in 3.14 on page 65
2. Filling in the notification fields of the structure allocated in the previous step
3. Calling the function *saNtfNotificationSend()* with the notification handle returned in step 1
4. Releasing the allocated memory with the *saNtfNotificationFree()* function

The second and third step may be repeated together multiple times, allowing for reuse of the allocated notification memory structure. Note that for subsequent uses of a notification structure, the number of elements in the arrays may be less, but must not be greater than the number that was specified with the allocate function. It is the responsibility of the Notification Service implementation to keep track of the number of array elements that once was allocated. Likewise, the used size of nested data section that were allocated with *saNtfPtrValAllocate* or *saNtfArrayValAllocate* may be less, but must not be greater than the size that was specified with the allocate function.

3.14.1 saNtfObjectCreateDeleteNotificationAllocate()

Prototype

```
SaAisErrorT saNtfObjectCreateDeleteNotificationAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfObjectCreateDeleteNotificationT *notification,
        SaUint16T numCorrelatedNotifications,
        SaUint16T lengthAdditionalText,
        SaUint16T numAdditionalInfo,
        SaUint16T numAttributes,
        SaInt16T variableDataSize
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notification - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numCorrelatedNotifications - [in] Number of correlated notifications in the notification

lengthAdditionalText - [in] Length of additional text in bytes (including terminating '\0')

numAdditionalInfo - [in] Number of additional info fields

numAttributes - [in] Number of object attributes in the notification

variableDataSize - [in] The maximum number of bytes that are to accommodate variable size notification data. In subsequent calls to the *saNtfPtrValAllocate()* and *saNtfArrayValAllocate()* functions, memory can be reserved up to *variableDataSize* for elements of a notification structure. Implementations of the Notification Service may use this size to preallocate memory in order to get PDU-ready notifications. Notification Service system limit is allocated if SA_NTF_ALLOC_SYSTEM_LIMIT is passed.

Description

This API internally allocates memory for an object create delete notification and initializes the *notification* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the notification structure. The pointers in *notification* are initialized to point to fields in the internal data structure. The returned *notification* structure also contains the notification handle that is used for subsequent calls of functions like *saNtfPtrValAllocate()*, *saNtfArrayValAllocate()*, *saNtfNotificationSend(),* and *saNtfNotificationFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_TOO_BIG - The variableDataSize is larger than the maximum permitted value.

See Also

saNtfInitialize(), saNtfNotificationSend(), saNtfNotificationFree()

3.14.2 saNtfAttributeChangeNotificationAllocate()

Prototype

```
SaAisErrorT saNtfAttributeChangeNotificationAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfAttributeChangeNotificationT *notification,
        SaUint16T numCorrelatedNotifications,
        SaUint16T lengthAdditionalText,
        SaUint16T numAdditionalInfo,
        SaUint16T numAttributes,
        SaInt16T variableDataSize
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notification - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numCorrelatedNotifications - [in] Number of correlated notifications in the notification

lengthAdditionalText - [in] Length of additional text in bytes (including terminating '\0')

numAdditionalInfo - [in] Number of additional info fields

numAttributes - [in] Number of changed attributes in the notification

variableDataSize - [in] The maximum number of bytes that are to accommodate variable size notification data. In subsequent calls to the *saNtfPtrValAllocate()* and *saNtfArrayValAllocate()* functions, memory can be reserved up to *variableDataSize* for elements of a notification structure. Implementations of the Notification Service may use this size to preallocate memory in order to get PDU-ready notifications. Notification Service system limit is allocated if SA_NTF_ALLOC_SYSTEM_LIMIT is passed.

Description

This API internally allocates memory for an attribute change notification and initializes the *notification* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notification* structure. The pointers in *notification* are initialized to point to fields in the internal data structure. The returned *notification* structure also contains the notification handle, which is used for subsequent calls of functions like *saNtfPtrValAllocate()*, *saNtfArrayValAllocate()*, *saNtfNotificationSend()* and *saNtfNotificationFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *notificationHandle* is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_TOO_BIG - The *variableDataSize* is larger than the maximum permitted value.

See Also

saNtfInitialize(), saNtfNotificationSend(), saNtfNotificationFree()

3.14.3 saNtfStateChangeNotificationAllocate()

Prototype

```
SaAisErrorT saNtfStateChangeNotificationAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfStateChangeNotificationT *notification,
        SaUint16T numCorrelatedNotifications,
        SaUint16T lengthAdditionalText,
        SaUint16T numAdditionalInfo,
        SaUint16T numStateChanges,
        SaInt16T variableDataSize
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notification - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numCorrelatedNotifications - [in] Number of correlated notifications in the notification

lengthAdditionalText - [in] Length of additional text in bytes (including terminating '\0')

numAdditionalInfo - [in] Number of additional info fields

numStateChanges - [in] Number of changed states in the notification

variableDataSize - [in] The maximum number of bytes that are to accommodate variable size notification data.In subsequent calls to the *saNtfPtrValAllocate()* and *saNtfArrayValAllocate()* functions, memory can be reserved up to *variableDataSize* for elements of a notification structure. Implementations of the Notification Service may use this size to preallocate memory in order to get PDU-ready notifications. Notification Service system limit is allocated if SA_NTF_ALLOC_SYSTEM_LIMIT is passed.

Description

This API internally allocates memory for a state change notification and initializes the *notification* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notification* structure. The pointers in *notification* are initialized to point to fields in the internal data structure. The returned *notification* structure also contains the notification handle, which is used for subsequent calls of functions like *saNtfPtrValAllocate()*, *saNtfArrayValAllocate()*, *saNtfNotificationSend()* and *saNtfNotificationFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_TOO_BIG - The variableDataSize is larger than the maximum permitted value.

See Also

saNtfInitialize(), saNtfNotificationSend(), saNtfNotificationFree()

3.14.4 saNtfAlarmNotificationAllocate()

Prototype

```
SaAisErrorT saNtfAlarmNotificationAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfAlarmNotificationT *notification,
        SaUint16T numCorrelatedNotifications,
        SaUint16T lengthAdditionalText,
        SaUint16T numAdditionalInfo,
        SaUint16T numSpecificProblems,
        SaUint16T numMonitoredAttributes,
        SaUint16T numProposedRepairActions,
        SaInt16T variableDataSize
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notification - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numCorrelatedNotifications - [in] Number of correlated notifications in the notification

lengthAdditionalText - [in] Length of additional text in bytes (including terminating '\0')

numAdditionalInfo - [in] Number of additional info fields

numSpecificProblems - [in] Number of specific problems

numMonitoredAttributes - [in] Number of monitored attributes

numProposedRepairActions - [in] Number of proposed repair actions

variableDataSize - [in] The maximum number of bytes that are to accommodate variable size notification data. In subsequent calls to the *saNtfPtrValAllocate()* and *saNtfArrayValAllocate()* functions, memory can be reserved up to *variableDataSize* for elements of a notification structure. Implementations of the Notification Service may use this size to preallocate memory in order to get PDU-ready notifications. Notification Service system limit is allocated if SA_NTF_ALLOC_SYSTEM_LIMIT is passed.

Description

This API internally allocates memory for an alarm notification and initializes the *notification* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notification* structure. The pointers in *notification* are initialized to point to fields in the internal data structure. The returned *notification* structure also contains the notification handle, which is used for subsequent calls of functions like *saNtfPtrValAllocate()*, *saNtfArrayValAllocate()*, *saNtfNotificationSend()* and *saNtfNotificationFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_TOO_BIG - The variableDataSize is larger than the maximum permitted value.

See Also

saNtfInitialize(), saNtfNotificationSend(), saNtfNotificationFree()

3.14.5 saNtfSecurityAlarmNotificationAllocate()

Prototype

```
SaAisErrorT saNtfSecurityAlarmNotificationAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfSecurityAlarmNotificationT *notification,
        SaUint16T numCorrelatedNotifications,
        SaUint16T lengthAdditionalText,
        SaUint16T numAdditionalInfo,
        SaInt16T variableDataSize
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notification - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numCorrelatedNotifications - [in] Number of correlated notifications in the notification

lengthAdditionalText - [in] Length of additional text in bytes (including terminating '\0')

numAdditionalInfo - [in] Number of additional info fields

variableDataSize - [in] The maximum number of bytes that are to accommodate variable size notification data. In subsequent calls to the *saNtfPtrValAllocate()* and *saNtfArrayValAllocate()* functions, memory can be reserved up to *variableDataSize* for elements of a notification structure. Implementations of the Notification Service may use this size to preallocate memory in order to get PDU-ready notifications. Notification Service system limit is allocated if SA_NTF_ALLOC_SYSTEM_LIMIT is passed.

Description

This API internally allocates memory for a security alarm notification and initializes the *notification* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notification* structure. The pointers in *notification* are initialized to point to fields in the internal data structure. The returned *notification* structure also contains the notification handle, which is used for subsequent calls of functions like *saNtfPtrValAllocate()*, *saNtfArrayValAllocate()*, *saNtfNotificationSend(),* and *saNtfNotificationFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_TOO_BIG - The variableDataSize is larger than the maximum permitted value.

See Also

saNtfInitialize(), saNtfNotificationSend(), saNtfNotificationFree()

3.14.6 saNtfPtrValAllocate()

Prototype

```
SaAisErrorT saNtfPtrValAllocate(
        SaNtfNotificationHandleT notificationHandle,
        SaUint16T dataSize,
        void **dataPtr,
        SaNtfValueT *value
);
```

Parameters

notificationHandle - [in] The handle, obtained through one of the *saNtf<notification type>NotificationAllocate()* functions, identifying the particular notification instance for which memory is to be reserved.

dataSize - [in] The number of bytes to be reserved.

dataPtr - [out] Pointer to the memory location that will be reserved with this function.

value - [in/out] An element of the notification structure has to be passed in, for which the function shall reserve memory. Implementations of the Notification Service are free to allocate memory in the preceding call to one of the *saNtf<notification type>NotificationAllocate()* functions or to allocate memory through this function. Memory allocated with this function is implicitly freed when *saNtfNotificationFree* is called. The offset and length of the reserved memory space is stored in *value*.

Description

This function reserves memory for an element of the notification structure in an internal structure and returns the pointer to the reserved memory region in *dataPtr*. This function may only be used with the *ptrVal* structure field in the *SaNtfValueT* union, i.e., for these data types: *SA_NTF_VALUE_LDAP_NAME, SA_NTF_VALUE_STRING, SA_NTF_VALUE_IPADDRESS and SA_NTF_VALUE_BINARY.*

The corresponding function in the Consumer API is *saNtfPtrValGet()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NO_SPACE - The requested memory cannot be reserved in the variable data area of the notification as there is not enough space left.

See Also

saNtfObjectCreateDeleteNotificationAllocate(), saNtfAttributeChangeNotificationAllocate(), saNtfStateChangeNotificationAllocate(), saNtfAlarmNotificationAllocate(), saNtfSecurityAlarmNotificationAllocate(),saNtfPtrValGet()

3.14.7 saNtfArrayValAllocate()

Prototype

```
SaAisErrorT saNtfArrayValAllocate(
        SaNtfNotificationHandleT notificationHandle,
        SaUint16T numElements,
        SaUint16T elementSize,
        void **arrayPtr,
        SaNtfValueT *value
);
```

Parameters

notificationHandle - [in] The handle, obtained through one of the *saNtf<notification type>NotificationAllocate()* functions, identifying the particular notification instance for which memory is to be reserved.

numElements - [in] Number of elements to be reserved.

elementSize - [in] Size of each element in the array in bytes.

arrayPtr - [out] Pointer to the memory location that will be reserved with this function.

value - [in/out] An element of the notification structure has to be passed in, for which the function shall reserve memory. Implementations of the Notification Service are free to allocate memory in the preceding call to one of the *saNtf<notification type>NotificationAllocate()* functions or to allocate memory through this function. Memory allocated with this function is implicitly freed when *saNtfNotificationFree* is called. The offset, size and field width of the memory reserved for the array is stored in *value*.

Description

This function reserves memory for an element of the notification structure of array type in an internal structure and returns the pointer to the reserved memory region in *dataPtr*. This function may only be used with the *arrayVal* structure field in the *SaNtfValueT* union, i.e., for the data type *SA_NTF_VALUE_ARRAY.*

The corresponding function in the Consumer API is *saNtfArrayValGet()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NO_SPACE - The requested memory cannot be reserved in the variable data area of the notification as there is not enough space left.

See Also

saNtfObjectCreateDeleteNotificationAllocate(), saNtfAttributeChangeNotificationAllocate(), saNtfStateChangeNotificationAllocate(), saNtfAlarmNotificationAllocate(), saNtfSecurityAlarmNotificationAllocate(), saNtfArrayValGet()

3.14.8 saNtfNotificationSend()

Prototype

SaAisErrorT saNtfNotificationSend(
SaNtfNotificationHandleT notificationHandle
)

Parameters

notificationHandle – [in] The handle, obtained through one of the *saNtf<notification type>NotificationAllocate()* functions, designating this particular notification instance.

Description

This method is used to send a notification. The notification is identified by the notification handle that is returned in the notification structure created with a preceding call to one of the *saNtf<notification type>NotificationAllocate()* functions.

The following table indicates which of the **header elements** in the notification referenced by *notificationHandle* are mandatory, optional, or implicit:

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
eventType	mandatory	–
notificationObject	mandatory	–
notifyingObject	optional	notificationObject
notificationClassId	mandatory	–
eventTime	optional – must be set to a valid timestamp or to SA_TIME_UNKNOWN	current system time (if SA_TIME_UNKNOWN is set)
notificationId	implicit	generated value
correlatedNotifications	optional – number of elements must be consistent with numCorrelatedNotifications	–
additionalText	optional – length must be consis-tent with lengthAdditionalText	–
additionalInfo	optional – number of elements must be consistent with numAdditionalInfo	–

In case *notificationHandle* refers to an **object create/delete notification**, the following table indicates which of the related elements are mandatory or optional:

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
notificationHandle	mandatory (returned by allocate function)	–
sourceIndicator	optional	SA_NTF_UNKNOWN_OPERATION
objectAttributes	optional – number of elements must be consistent with numAttributes	–

In case *notificationHandle* refers to an **attribute value change notification**, the following table indicates which of the related elements are mandatory or optional:

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
notificationHandle	mandatory (returned by allocate function)	–
sourceIndicator	optional	SA_NTF_UNKNOWN_OPERATION
changedAttributes	mandatory (the sub-element oldAttributeValue is optional)	–

In case *notificationHandle* refers to a **state change notification**, the following table indicates which of the related elements are mandatory or optional:

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
notificationHandle	mandatory (returned by allocate function)	–
sourceIndicator	optional	SA_NTF_UNKNOWN_OPERATION
changedStates	mandatory (the sub-element oldState is optional)	–

In case *notificationHandle* refers to an **alarm notification**, the following table indicates which of the related elements are mandatory or optional:

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
notificationHandle	mandatory (returned by allocate function)	–
probableCause	mandatory	–
specificProblems	optional – number of elements must be consistent with numSpecificProblems	–
perceivedSeverity	mandatory	–
trend	optional	SA_NTF_TREND_NO_CHANGE
thresholdInformation	optional	–
monitoredAttributes	optional – number of elements must be consistent with numMonitoredAttributes	–
proposedRepairActions	optional – number of elements must be consistent with numProposedRepairActions	–

In case *notificationHandle* refers to a **security alarm notification**, the following table indicates which of the related elements are mandatory or optional:

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
notificationHandle	mandatory (returned by allocate function)	–
probableCause	mandatory	–
severity	mandatory	–
securityAlarmDetector	mandatory	–
serviceUser	mandatory	–
serviceProvider	mandatory	–

If successful, the notification identifier is written to the field pointed to by *notificationId* in the *SaNtfNotificationHeaderT* part of the notification referenced by *notificationHandle*. It can later be used for referencing preceding notification instances.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfObjectCreateDeleteNotificationAllocate(), saNtfAttributeChangeNotificationAllocate(), saNtfStateChangeNotificationAllocate(), saNtfAlarmNotificationAllocate(), saNtfSecurityAlarmNotificationAllocate()

3.14.9 saNtfNotificationFree()

Prototype

SaAisErrorT saNtfNotificationFree(
SaNtfNotificationHandleT notificationHandle
)

Parameters

notificationHandle – [in] The handle, obtained through one of the *saNtf<notification type>NotificationAllocate()* functions, designating this particular notification instance.

Description

Frees the memory previously allocated for a notification.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

See Also

saNtfObjectCreateDeleteNotificationAllocate(),
saNtfAttributeChangeNotificationAllocate(),
saNtfStateChangeNotificationAllocate(),saNtfAlarmNotificationAllocate(),
saNtfSecurityAlarmNotificationAllocate(), saNtfNotificationSend()

3.15 Consumer Operations

3.15.1 Filtering

There are functions to allocate filters specific for all notification types. Functions of the Subscriber API and Reader API take a *SaNtfNotificationTypeFilterHandlesT* parameter that contains handles for all notification type specific filters. (Filter handles are returned by the filter allocation functions.) For each notification type the caller is interested in filter criteria have to be specified (i.e., a filter handle has to be set in the *SaNtfNotificationTypeFilterHandlesT* parameter). If the caller is not interested in notifications of a particular type the special handle *SA_NTF_FILTER_HANDLE_NULL* has to be set.

When allocating a filter for a particular notification type, several filter elements may be specified. For instance, for alarm notifications there are filter elements for Probable Cause, Perceived Severity etc. Each such filter element can be a set of explicit values or an empty set. If not empty, then the filter element matches for a particular notification if one of the specified explicit values match. This means that for one filter element all values are logically ORed. If a filter element is an empty set, then all values of a notification for that particular element match (pass through). All filter elements for a notification type are logically ANDed. This means that a notification matches the filter if it matches all filter elements, respectively.

When filtering is done, in most cases the values of a filter element are checked for equality against the related value of a notification. This can be called *low level filtering*, since there is no interpretation done on the meaning of a particular value of a filter element. However, for Notification Object and Notifying Object filter elements there is also *high level filtering* provided in these two specific cases:

- If the value of a filter element contains the LDAP DN of an AMF service unit, then any AMF component belonging to that service unit matches.
- Likewise, if the value of a filter element contains the LDAP DN of an AMF service instance, then any AMF component service instance belonging to that service instance matches that filter element.

Note that the above described filtering for Notification Object and Notifying Object does *not* create a dependency on the AMF service. The AMF LDAP DNs directly contains the information about the relationship between service units and components as well as that between service instances and component service instances. This is because the DN of a component has the full DN of the containing service unit and the DN of a component service instance has the full DN of the containing service instance in it.

The same notification filters can be used for multiple reads or subscriptions. It is the responsibility of the process to free the notification filters by invoking the

saNtfNotificationFilterFree() function if the notification filters are no longer needed after calls to functions of the Subscriber or Reader API.

3.15.2 Common Consumer Operations

This section contains functions, which are common to the subscriber and reader interface. This comprises a function to retrieve the localized notification message text, functions to access the contents of nested notification elements and filter allocation functions for the various notification types.

3.15.2.1 saNtfLocalizedMessageGet()

Prototype

```
SaAisErrorT saNtfLocalizedMessageGet(
        SaNtfNotificationHandleT notificationHandle,
        SaStringT *message
);
```

Parameters

notificationHandle - [in] notification handle

message – [out] – A pointer to the buffer for the localized message that is to be returned. Message may not be NULL. If the function returns successfully then *message points to memory allocated by the function, which contains the localized message. The calling application is responsible to free this memory when it is no longer needed using *saNtfLocalizedMessageFree()*. If the function returns an error then there has no memory been allocated for *message.

Description

Returns a localized textual description of the situation that resulted in the notification referenced by the given notification handle. The localized message text consists of UTF-8 encoded characters and is terminated by the '\0' character.

This function is intended to be used after a notification has been retrieved either by the *SaNtfNotificationCallbackT* callback or *saNtfNotificationReadNext*. If there are no localization data available for the notification class identifier in the notification referenced by *notificationHandle* this function returns *SA_AIS_ERR_NOT_EXIST*. Localization data are optional and need not be provided for all notification class identifiers. In case an implementation of the Notification Service does not support internationalization this function returns *SA_AIS_ERR_NOT_SUPPORTED*.

The format string of localization data related to the notification may contain references to data elements in the notification. In the returned message usually these

references are replaced by the referenced values. In case the referenced value is not contained in the notification the returned message keeps the reference as it is in the format string. Refer to 4.2 on page 119 for details about the format string.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_NOT_EXIST – There is no localization data available for the notification class identifier in the notification referenced by notificationHandle.

SA_AIS_ERR_NOT_SUPPORTED – The implementation of the Notification Service does not support the optional functionality of internationalization.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

SA_AIS_ERR_NO_MEMORY - Either the service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

SaNtfNotificationCallbackT, saNtfLocalizedMessageFree(), saNtfNotificationSubscribe(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext()

3.15.2.2 saNtfLocalizedMessageFree()

Prototype

```
SaAisErrorT saNtfLocalizedMessageFree(
        SaStringT message
)
```

Parameters

message – [in] The message buffer, obtained through the *saNtfLocalizedMessageGet()* function.

Description

Frees the memory previously allocated for a localized message by *saNtfLocalizedMessageGet()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

See Also

saNtfLocalizedMessageGet()

3.15.2.3 saNtfPtrValGet()

Prototype

```
SaAisErrorT saNtfPtrValGet(
        SaNtfNotificationHandleT notificationHandle,
        const SaNtfValueT *value,
        void **dataPtr,
        SaUint16T *dataSize
);
```

Parameters

notificationHandle - [in] The notification handle is obtained through the notification structure passed to one of the callbacks of the Subscriber API or returned from one of the functions of the Reader API.

value - [in] Element of the notification structure, the data of which is to be returned in *dataPtr*. The offset and length of the reserved memory space is taken from *value*.

dataPtr - [out] Pointer to the returned data. Since the returned pointer points to the internal data structure it is no longer valid after the enclosing notification has been freed with *saNtfNotificationFree()*.

dataSize - [out] The size of the data associated with the given *value*.

Description

This function gets the pointer to a memory location allocated in an internal structure associated with the notification instance that is reserved for the given element of the notification structure. It may only be used with the *ptrVal* structure in the *SaNtfValueT* union.

This function is the counterpart of *saNtfPtrValAllocate()* in the Producer API.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

See Also

saNtfObjectCreateDeleteNotificationAllocate(), saNtfAttributeChangeNotificationAllocate(), saNtfStateChangeNotificationAllocate(), saNtfAlarmNotificationAllocate(), saNtfSecurityAlarmNotificationAllocate(), saNtfPtrValAllocate()

3.15.2.4 saNtfArrayValGet()

Prototype

```
SaAisErrorT saNtfArrayValGet(
        SaNtfNotificationHandleT notificationHandle,
        const SaNtfValueT *value,
        void **arrayPtr,
        SaUint16T *numElements,
        SaUint16T *elementSize
);
```

Parameters

notificationHandle - [in] The notification handle is obtained through the notification structure passed to one of the callbacks of the Subscriber API or returned from one of the functions of the Reader API.

value - [in] Element of notification structure, the data of which is to be returned in *arrayPtr*. The offset, size and field width of the reserved array space is taken from *value*.

arrayPtr - [out] Pointer to the returned array. Since the returned pointer points to the internal data structure it is no longer valid after the enclosing notification has been freed with *saNtfNotificationFree()*.

numElements - [out] Number of elements in the array.

elementSize - [out] Size of each element in the array.

Description

This function gets the pointer to a memory location allocated in an internal structure associated with the notification instance that is reserved for the given array type element of the notification structure. It may only be used with the *arrayVal* structure field in the *SaNtfValueT* union.

This function is the counterpart of *saNtfArrayValAllocate()* in the Producer API.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

See Also

saNtfObjectCreateDeleteNotificationAllocate(), saNtfAttributeChangeNotificationAllocate(), saNtfStateChangeNotificationAllocate(), saNtfAlarmNotificationAllocate(), saNtfSecurityAlarmNotificationAllocate(), saNtfArrayValAllocate()

3.15.2.5 saNtfObjectCreateDeleteNotificationFilterAllocate()

Prototype

```
SaAisErrorT saNtfObjectCreateDeleteNotificationFilterAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfObjectCreateDeleteNotificationFilterT *notificationFilter,
        SaUint16T numEventTypes,
        SaUint16T numNotificationObjects,
        SaUint16T numNotifyingObjects,
        SaUint16T numNotificationClassIds,
        SaUint16T numSourceIndicators
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notificationFilter - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numEventTypes - [in] Number of event types in the notification filter

numNotificationObjects - [in] Number of notification objects in the notification filter

numNotifyingObjects - [in] Number of notifying objects in the notification filter

numNotificationClassIds - [in] Number of notification class IDs in the notification filter

numSourceIndicators - [in] Number of source indicators in the notification filter

Description

This API internally allocates memory for an object create delete notification filter and initializes the *notificationFilter* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notificationFilter* structure. The pointers in *notificationFilter* are initialized to point to fields in the internal data structure. The returned *notificationFilter* structure also contains the notification filter handle, which is used for subsequent calls of functions like *saNtfNotificationSubscribe()*, *saNtfNotificationReadInitialize()*, saNtfNotificationReadNext*()* or *saNtfNotificationFilterFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfInitialize(), saNtfNotificationSubscribe(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext(), saNtfNotificationFilterFree()

3.15.2.6 saNtfAttributeChangeNotificationFilterAllocate()

Prototype

```
SaAisErrorT saNtfAttributeChangeNotificationFilterAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfAttributeChangeNotificationFilterT *notificationFilter,
        SaUint16T numEventTypes,
        SaUint16T numNotificationObjects,
        SaUint16T numNotifyingObjects,
        SaUint16T numNotificationClassIds,
        SaUint16T numSourceIndicators
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notificationFilter - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numEventTypes - [in] Number of event types in the notification filter

numNotificationObjects - [in] Number of notification objects in the notification filter

numNotifyingObjects - [in] Number of notifying objects in the notification filter

numNotificationClassIds - [in] Number of notification class IDs in the notification filter

numSourceIndicators - [in] Number of source indicators in the notification filter

Description

This API internally allocates memory for an attribute change notification filter and initializes the *notificationFilter* structure. The returned *notificationFilter* structure also contains the notification filter handle, which is used for subsequent calls of functions like *saNtfNotificationSubscribe()*, *saNtfNotificationReadInitialize()*, *saNtfNotificationReadNext(),* or *saNtfNotificationFilterFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfInitialize(), saNtfNotificationSubscribe(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext(), saNtfNotificationFilterFree()

3.15.2.7 saNtfStateChangeNotificationFilterAllocate()

Prototype

```
SaAisErrorT saNtfStateChangeNotificationFilterAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfStateChangeNotificationFilterT *notificationFilter,
        SaUint16T numEventTypes,
        SaUint16T numNotificationObjects,
        SaUint16T numNotifyingObjects,
        SaUint16T numNotificationClassIds,
        SaUint16T numSourceIndicators,
        SaUint16T numChangedStates
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notificationFilter - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numEventTypes - [in] Number of event types in the notification filter

numNotificationObjects - [in] Number of notification objects in the notification filter

numNotifyingObjects - [in] Number of notifying objects in the notification filter

numNotificationClassIds - [in] Number of notification class IDs in the notification filter

numSourceIndicators - [in] Number of source indicators in the notification filter

numChangedStates - [in] Number of changed states in the notification filter

Description

This API internally allocates memory for an attribute change notification filter and initializes the *notificationFilter* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notificationFilter* structure. The pointers in *notificationFilter* are initialized to point to fields in the internal data structure. The returned *notificationFilter* structure also contains the notification filter handle, which is used for subsequent calls of functions like *saNtfNotificationSubscribe()*, *saNtfNotificationReadInitialize()*, *saNtfNotificationReadNext(),* or *saNtfNotificationFilterFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfInitialize(), saNtfNotificationSubscribe(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext(), saNtfNotificationFilterFree()

3.15.2.8 saNtfAlarmNotificationFilterAllocate()

Prototype

```
SaAisErrorT saNtfAlarmNotificationFilterAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfAlarmNotificationFilterT *notificationFilter,
        SaUint16T numEventTypes,
        SaUint16T numNotificationObjects,
        SaUint16T numNotifyingObjects,
        SaUint16T numNotificationClassIds,
        SaUint16T numProbableCauses,
        SaUint16T numPerceivedSeverities,
        SaUint16T numTrends
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notificationFilter - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numEventTypes - [in] Number of event types in the notification filter.

numNotificationObjects - [in] Number of notification objects in the notification filter.

numNotifyingObjects - [in] Number of notifying objects in the notification filter.

numNotificationClassIds - [in] Number of notification class IDs in the notification filter.

numProbableCauses - [in] Number of probable causes in the notification filter.

numPerceivedSeverities - [in] Number of perceived severities in the notification filter.

numTrends - [in] Number of trends in the notification filter.

Description

This API internally allocates memory for an alarm notification filter and initializes the *notificationFilter* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notificationFilter* structure. The pointers in *notificationFilter* are initialized to point to fields in the internal data structure. The returned *notificationFilter* structure also contains the notification filter handle, which is used for subsequent calls of functions like *saNtfNotificationSubscribe()*, *saNtfNotificationReadInitialize()*, *saNtfNotificationReadNext(),* or *saNtfNotificationFilterFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfInitialize(), saNtfNotificationSubscribe(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext(), saNtfNotificationFilterFree()

3.15.2.9 saNtfSecurityAlarmNotificationFilterAllocate()

Prototype

```
SaAisErrorT saNtfSecurityAlarmNotificationFilterAllocate(
        SaNtfHandleT ntfHandle,
        SaNtfSecurityAlarmNotificationFilterT *notificationFilter,
        SaUint16T numEventTypes,
        SaUint16T numNotificationObjects,
        SaUint16T numNotifyingObjects,
        SaUint16T numNotificationClassIds,
        SaUint16T numProbableCauses,
        SaUint16T numSeverities,
        SaUint16T numSecurityAlarmDetectors,
        SaUint16T numServiceUsers,
        SaUint16T numServiceProviders
);
```

Parameters

ntfHandle - [in] The handle, obtained through the *saNtfInitialize()* function, designating this particular initialization of the Notification Service.

notificationFilter - [out] This variable can be on the stack or heap, i.e., it has to be allocated by the invoking process.

numEventTypes - [in] Number of event types in the notification filter

numNotificationObjects - [in] Number of notification objects in the notification filter

numNotifyingObjects - [in] Number of notifying objects in the notification filter

numNotificationClassIds - [in] Number of notification class IDs in the notification filter

numProbableCauses - [in] Number of probable causes in the notification filter

numSeverities - [in] Number of severities in the notification filter

numSecurityAlarmDetectors - [in] Number of security alarm detectors in the notification filter

numServiceUsers - [in] Number of service users in the notification filter

numServiceProviders - [in] Number of service providers in the notification filter

Description

This API internally allocates memory for a security alarm notification filter and initializes the *notificationFilter* structure. The values of the function parameters indicating a length or the number of array elements are copied to the related attributes in the *notificationFilter* structure. The pointers in *notificationFilter* are initialized to point to fields in the internal data structure. The returned *notificationFilter* structure also contains the notification filter handle, which is used for subsequent calls of functions like *saNtfNotificationSubscribe()*, *saNtfNotificationReadInitialize()*, *saNtfNotificationReadNext(),* or *saNtfNotificationFilterFree()*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfInitialize(), saNtfNotificationSubscribe(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext(), saNtfNotificationFilterFree()

3.15.2.10 saNtfNotificationFilterFree()

Prototype

SaAisErrorT saNtfNotificationFilterFree(
SaNtfNotificationFilterHandleT notificationFilterHandle
)

Parameters

notificationFilterHandle – [in] notification filter handle

Description

Frees the memory previously allocated for a notification filter.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle notificationFilterHandle is invalid, since it is corrupted, uninitialized, or has already been freed.

See Also

saNtfObjectCreateDeleteNotificationFilterAllocate(), saNtfAttributeChangeNotificationFilterAllocate(), saNtfStateChangeNotificationFilterAllocate(), saNtfAlarmNotificationFilterAllocate(), saNtfSecurityAlarmNotificationFilterAllocate()

3.15.3 Operations of the Subscriber API

This section describes the API functions that enable the caller to receive notifications as they occur. The procedure for receiving notifications is divided into several steps:

1. Registering at the Notification Service with the *saNtfInitialize()* function and supplying callback functions for handling received notifications
2. Allocating memory for the notification filter contents with one or several of the allocation functions described in *Filtering* on page 85
3. Filling in the notification filter fields of the structure or structures allocated in the previous step
4. *Calling saNtfNotificationSubscribe()* with the filter handles returned in step 2
5. Releasing the allocated memory with the *saNtfNotificationFilterFree()* function

Steps 3 and 4 may be repeated multiple times for reuse of the allocated notification filter structures. Note that for subsequent uses of a filter structure, the number of elements in the arrays may be less, but must not be greater than the number that was specified with the allocate function. It is the responsibility of the Notification Service implementation to keep track about the number of array elements that once was allocated.

3.15.3.1 saNtfNotificationSubscribe()

Prototype

```
SaAisErrorT saNtfNotificationSubscribe(
        const SaNtfNotificationTypeFilterHandlesT *notificationFilterHandles,
        SaNtfSubscriptionIdT subscriptionId
);
```

Parameters

notificationFilterHandles - [in] A pointer to the handles of the notification type specific filters previously returned by the alloc functions. At least for one notification type a filter must have been allocated. If more than one handle is used in the structure, then all handles must have been generated for the same instance of the Notification Service (i.e., with the same *SaNtfHandleT* value). Notification types for which no subscription is to be made must have set their corresponding field in *notificationFilterHandles* to *SA_NTF_FILTER_HANDLE_NULL*.

subscriptionId – [in] Used to identify a particular subscription within the context of the subscriber application. It must be unique for all subscriptions made for the instance of the Notification Service, which is indirectly referenced by all handles set in *notificationFilterHandles*. It is also passed to subsequent invocations of the notification callback. In the context of the notification callback it is useful when the

application has made several subscriptions. The subscription ID has to be used when unsubscribing with *saNtfNotificationUnsubscribe*.

Description

The *saNtfNotificationSubscribe()* function enables a process to subscribe for notifications by registering one or more filters referenced by *notificationFilterHandles*.

Notifications are delivered via the invocation of the notification type specific callback function, which must have been supplied when the process called the *saNtfInitialize()* function.

This function consumes the filters. After a call to this function, the process may safely free the filters with *saNtfNotificationFilterFree()* or use them for other calls of the Consumer APIs (i.e., *saNtfNotificationReadInitialize* or *saNtfNotificationSubscribe()*). It is the responsibility of the process to free the notification filters by invoking the *saNtfNotificationFilterFree()* function if the notification filters are no longer needed.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE – One of the handles in *notificationFilterHandles* is invalid, since it is corrupted, uninitialized, or has already been freed or not all handles refer to the same instance of the Notification Service.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_EXIST – A subscription with the value of *subscriptionId* already exists for this instance of the Notification Service (i.e., the *SaNtfHandleT* value that was used to allocate the filters).

See Also

saNtfInitialize(), saNtfDispatch(), saNtfNotificationUnsubscribe(), saNtfObjectCreateDeleteNotificationFilterAllocate(), saNtfAttributeChangeNotificationFilterAllocate(), saNtfStateChangeNotificationFilterAllocate(), saNtfAlarmNotificationFilterAllocate(), saNtfSecurityAlarmNotificationFilterAllocate()

3.15.3.2 saNtfNotificationUnsubscribe()

Prototype

```
SaAisErrorT saNtfNotificationUnsubscribe(
        SaNtfSubscriptionIdT subscriptionId
);
```

Parameters

subscriptionId – [in] Subscription identifier, which was passed to *saNtfNotificationSubscribe* before.

Description

The *saNtfNotificationUnsubscribe()* function deletes the subscription previously made with a call to *saNtfNotificationSubscribe()*.

Queued notifications matching the filter settings passed at subscription time and that have not been delivered to the subscriber by one of the notification callbacks are implicitly discarded. If the subscriber wants to avoid this it must make a new subscription before it deletes the old one.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_EXIST – A subscription with the value of *subscriptionId* does not exist for this instance of the Notification Service (i.e., the *SaNtfHandleT* value, which was used to allocate the filters used for the subscription).

See Also

saNtfNotificationSubscribe()

3.15.3.3 SaNtfNotificationCallbackT

Prototype

```
typedef void (*SaNtfNotificationCallbackT)(
        SaNtfSubscriptionIdT subscriptionId,
        const SaNtfNotificationsT *notification
);
```

Parameters

subscriptionId – [in] An identifier that the subscriber supplied in a *saNtfNotificationSubscribe()* invocation that enables the subscriber to determine which subscription resulted in the delivery of the notification.

notification - [in] The notification delivered by this callback.

Description

The Notification Service invokes this callback function to deliver a notification to the subscriber. This callback is invoked in the context of a thread issuing a s*aNtfDispatch()* call.

It is the responsibility of the process to free the notification by invoking the *saNtfNotificationFree()* function.

Return Values

None.

See Also

saNtfNotificationSubscribe(), saNtfNotificationFree(), saNtfDispatch()

3.15.3.4 SaNtfNotificationDiscardedCallbackT

Prototype

```
typedef void (*SaNtfNotificationDiscardedCallbackT)(
        SaNtfSubscriptionIdT subscriptionId,
        SaNtfNotificationTypeT notificationType,
        SaUint32T numberDiscarded,
        const SaNtfIdentifierT *discardedNotificationIdentifiers
);
```

Parameters

subscriptionId – [in] An identifier that a process supplied in a *saNtfNotificationSubscribe()* invocation that enables it to determine for which subscription notifications have been discarded.

notificationType - [in] The notification type of the discarded notifications.

numberDiscarded - [in] The number of discarded notifications.

discardedNotificationIdentifiers - [in] The list of notification identifiers of the discarded notifications. For notification types other than SA_NTF_TYPE_ALARM or SA_NTF_TYPE_SECURITY_ALARM this pointer may be NULL. If not NULL this array contains as many elements as indicated by *numberDiscarded*.

Description

The Notification Service invokes this callback function to notify a subscriber that one or more notifications of a particular notification type have been discarded. This callback is invoked in the context of a thread issuing a *saNtfDispatch()* call. Unless *discardedNotificationIdentifiers* is NULL the subscriber can use the Reader API to retrieve the notifications.

If the subscriber wants to keep the contents of *discardedNotificationIdentifiers* for processing after it has returned from this callback it has to make a copy of it before returning. *discardedNotificationIdentifiers* and its contents are fully controlled by the Notification Service library. In particular, the subscriber must not change the contents of or call *free()* for *discardedNotificationIdentifiers*.

Refer also to „Discarded Notifications“

Return Values

None.

See Also

saNtfNotificationSubscribe(), saNtfDispatch(), saNtfNotificationReadInitialize(), saNtfNotificationReadNext()

3.15.4 Operations of the Reader API

This section describes the API functions, which enable the caller to read logged notifications. Reading logged notifications is divided into several steps:

1. Allocating memory for the notification filter contents with one or several of the allocation functions in *Filtering* on page 85
2. Filling in the notification filter fields of the structure or structures allocated in the previous step
3. Calling *saNtfNotificationReadInitialize()* with the filter handles returned from step 1 to obtain a read handle
4. Releasing the memory allocated for the filters with the *saNtfNotificationFilterFree()* function (if not needed otherwise)
5. Calling *saNtfNotificationReadNext()* with the read handle returned from step 3 and specifying the search direction
6. Releasing the memory allocated for the read handle with the *saNtfNotificationReadFinalize()* function

Step 5 may be repeated multiple times.

3.15.4.1 saNtfNotificationReadInitialize()

Prototype

```
SaAisErrorT saNtfNotificationReadInitialize(
        SaNtfSearchCriteriaT searchCriteria,
        const SaNtfNotificationTypeFilterHandlesT *notificationFilterHandles,
        SaNtfReadHandleT *readHandle
);
```

Parameters

searchCriteria – [in] In addition to the filter criteria, this parameter may specify that the search should be started based on event time or notification identifier. If the search mode is set to SA_NTF_SEARCH_ONLY_FILTER then the search will start with the oldest notification that exists.

notificationFilterHandles - [in] A pointer to the handles of the notification type specific filters previously generated by the filter allocate functions. At least for one notification type a filter must have been allocated. If more than one handle is used in the structure, then all handles must have been generated for the same instance of the Notification Service (i.e., with the same *SaNtfHandleT* value). Notification types that are not to be read must have set their corresponding field in *notificationFilterHandles* to *SA_NTF_FILTER_HANDLE_NULL*.

readHandle – [out] The read handle returned by the function. The read handle is to be used for subsequent calls of *saNtfNotificationReadNext*(). When no more notifications are to be read for the given filter criteria the application must free its related resources with *saNtfNotificationReadFinalize()*.

Description

This method is used to initialize reading logged notifications according to the search criteria specified by *searchCriteria* and filters referenced by *notificationFilterHandles*.

The search criteria are optional. They can be used to specify a point in time or a notification identifier where reading of notifications should start when *saNtfNotificationReadNext*() is called the first time with the new read handle. The typical way to read a chronologically ordered list of notifications is to specify a starting point and filter criteria with *saNtfNotificationReadInitialize()* and to read notifications with a sequence of calls to *saNtfNotificationReadNext()*.

The filters referenced by *notificationFilterHandles* may specify additional filter criteria. For each notification type the caller is interested in filter criteria have to be specified (i.e., a filter handle has to be set in *notificationFilterHandles*). An implementation need not support notification types other than alarm notifications and security alarm notifications in this function. If a reader application sets a filter handle for a notification type that is not supported by the implementation, *SA_AIS_ERR_NOT_SUPPORTED* is returned.

This function consumes the filters. After a call to this function the process may safely free the filters with *saNtfNotificationFilterFree()* or use them for other calls of the Consumer APIs (i.e., *saNtfNotificationReadInitialize* or *saNtfNotificationSubscribe()*). It is the responsibility of the process to free the notification filters by invoking the *saNtfNotificationFilterFree()* function if the notification filters are no longer needed.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE – One of the handles in *notificationFilterHandles* is invalid, since it is corrupted, uninitialized, or has already been freed or not all handles refer to the same instance of the Notification Service.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NOT_SUPPORTED – In *notificationFilterHandles* at least one handle has been set for a notification type, which is not supported by this implementation.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saNtfNotificationReadNext(), saNtfNotificationReadFinalize(), saNtfObjectCreateDeleteNotificationFilterAllocate(), saNtfAttributeChangeNotificationFilterAllocate(), saNtfStateChangeNotificationFilterAllocate(), saNtfAlarmNotificationFilterAllocate(), saNtfSecurityAlarmNotificationFilterAllocate()

3.15.4.2 saNtfNotificationReadNext()

Prototype

```
SaAisErrorT saNtfNotificationReadNext(
        SaNtfReadHandleT readHandle,
        SaNtfSearchDirectionT searchDirection,
        SaNtfNotificationsT *notification
);
```

Parameters

readHandle – [in] The read handle, which was previously obtained by a call to *saNtfNotificationReadInitialize()*.

searchDirection – [in] Indicates if the notification to be read should be in ascending (*SA_NTF_SEARCH_OLDER)* or descending (*SA_NTF_SEARCH_YOUNGER*) chronological order with respect to the previously read notification. For the first invocation of this function after *saNtfNotificationReadInitialize(),* this parameter is ignored.

notificationFilterHandles - [in] A pointer to the handles of the notification type specific filters previously generated by the alloc functions. At least for one notification type a filter must have been allocated. If more than one handle is used in the structure, then all handles must have been generated for the same instance of the Notification Service (i.e., with the same *SaNtfHandleT* value). Notification types which are not to be read must have set their corresponding field in *notificationFilterHandles* to *SA_NTF_FILTER_HANDLE_NULL*.

notification – [out] The notification returned by the function. The *notificationType* field determines which of the fields in the union is valid, i.e., which field actually contains the notification. This variable can be on the stack or heap, i.e. it has to be allocated by the invoking process. After the notification is no longer used the application must free its related resources with *saNtfNotificationFree()*.

Description

This method is used to read chronologically ordered logged notifications. Reading must have been initialized with *saNtfNotificationReadInitialize()*. As many as desired notifications may then be read with a sequence of calls to *saNtfNotificationReadNext()*.

When this function is called the first time after the read handle has been obtained from *saNtfNotificationReadInitialize()* it will ignore the *searchDirection* parameter and will use only the search and filter criteria passed to *saNtfNotificationReadInitialize().* For subsequent invocations with the same read handle, this function uses

searchDirection and the filter criteria previously passed to *saNtfNotificationReadInitialize()*.

If successful, a call to this function stores internally context information about the found notification. The context information is bound to the read handle. In a subsequent call to *saNtfNotificationReadNext(),* this context information is used to retrieve the chronologically next (or previous) notification.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE – *readHandle* is invalid, since it is corrupted, uninitialized, or has already been freed.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Notification Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_EXIST – There is no notification matching the given criteria.

See Also

saNtfNotificationReadInitialize(), saNtfNotificationReadFinalize(), saNtfObjectCreateDeleteNotificationFilterAllocate(), saNtfAttributeChangeNotificationFilterAllocate(),

saNtfStateChangeNotificationFilterAllocate(), saNtfAlarmNotificationFilterAllocate(), saNtfSecurityAlarmNotificationFilterAllocate()

3.15.4.3 saNtfNotificationReadFinalize()

Prototype

```
SaAisErrorT saNtfNotificationReadFinalize(
        SaNtfReadHandleT readHandle
);
```

Parameters

readHandle – [in] The read handle previously obtained by *saNtfNotificationReadInitialize*()

Description

This function is used to release any resources bound to the passed read handle. The read handle may no longer be used for calls to *saNtfNotificationReadNext().*

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it did not.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE – *readHandle* is invalid, since it is corrupted, uninitialized, or has already been freed.

See Also

saNtfNotificationReadNext(), saNtfNotificationReadInitialize(), saNtfObjectCreateDeleteNotificationFilterAllocate(), saNtfAttributeChangeNotificationFilterAllocate(), saNtfStateChangeNotificationFilterAllocate(), saNtfAlarmNotificationFilterAllocate(), saNtfSecurityAlarmNotificationFilterAllocate()

3.16 Notification Suppression

This is the description of the suppression API of the Notification Service. Currently, the Notification service does not provide an explicit mechanism for suppressing notifications. Such a mechanism is expected to be included in a future version of this specification. It is possible that the future specification of a suppression mechanism will potentially involve administrative as well as runtime API specification to enable suppression.This section examines certain suppression mechanisms that could be possibly integrated in a future release.

The mechanism of notification suppression is essential to avoid situations where floods of unimportant/dispensable notifications are generated. These are typical reasons why it makes sense to have an administrative API to suppress notifications:

- Notification floods may contain many notifications reflecting only tiny changes in the system or repeated notifications informing about the same situation.
- If the structure of managed objects or object attributes is fine-grained and a notification is generated for each object creation and deletion and attribute value change, a notification flood may be the result.
- Errors in the programming logic of a software package may cause repeated notifications that inform about the same situation. As absolutely bug-free software is an illusion, the Notification Service has to protect itself and its subscribers (and thus the human end user) against uses of the producer API that would result in a flood of notifications.
- Some of the object create/delete or attribute value change notifications could be quite important when system integration and test takes place but could be of little interest on a productive system.

The above described situations for notification floods need different handling: For example in the first case, suppression of all notifications about those insignificant changes may be needed, in the second case, the first notification informing about a noteworthy situation must certainly be generated but all subsequent notifications about the same situation should be suppressed.

Notification floods would overload the Notification Service, burden a notification subscriber program with additional filtering of the received notifications and burden a human operator, who is responsible for monitoring the notifications, with the additional work of extracting the important information from the flood of notifications.

There are two different types of suppression:

- *Static suppression*
  No notification matching the suppression filter criteria for static suppression will be forwarded to subscribers or logged.

- *Dynamic suppression*
  For each produced notification matching the filter criteria for dynamic suppression, a maximum number of instances per time interval is not exceeded. For instance, 2 instances of a particular notification produced within 60 seconds would be forwarded and logged as usual, but more instances within the same time interval would be suppressed.

The Notification Service provides close-to-source suppression. This is important for the suppression to be most efficient. Close-to-source suppression has two aspects:

- If a producer knows that particular notifications are currently suppressed, the producer can save even those efforts necessary to construct a currently suppressed notification.
- The producer API part of the notification library suppresses notifications matching the current suppression settings.

Notification suppression is only possible for object create/delete notifications, attribute value change notifications and state change notifications. It is not possible to suppress alarms and security alarms. As an optimization an implementation of the Notification Service may implicitly suppress notifications when both of the following conditions are met:

- The implementation of the Notification Service does not provide logging for the notification type, i.e., the notification type is one of object creation / deletion, attribute value change or state change.
- There is currently no subscription for this notification.

4 Configuration

This chapter describes the data to be configured for the Notification Service. The description is high level. A XML description is still missing.

4.1 Trap OID Mapping

For the SNMP interface, it is important that Notification Class Identifiers (NCIs) can be mapped to OIDs of SNMP traps. This will allow a SNMP manager to easily identify notifications sent as SNMP traps.

The mapping table is read by the SNMP agent, which acts as a notification subscriber and needs to find the related trap OID for an incoming notification. It does a table lookup using the NCI as index into the mapping table.

Field Name	Data Type	Description
NCI	SaNtfClassIdT	The Notification Class Identifier; index into this table (= unique key). The value {0, 0, 0} has a special meaning: If existing, it identifies the mapping which is used by default when no record is found for a particular NCI.
Trap OID	string	OID of the SNMP trap in dot-notation.

SA Forum defines mappings for those NCIs specified by SA Forum. Specific AIS implementations and vendor-specific applications may add mappings for their own notification classes.

4.2 Internationalization

Optionally, for each NCI internationalized textual information can be configured. This information refers to localized message catalogs which can be installed on a system for multiple languages. Currently, POSIX and GNU message catalogs are supported. Entries for both catalog types can be freely intermixed in the table. However, for one particular NCI there can be only an entry for one catalog type.

For POSIX message catalogs these configuration data fields are relevant:

- Catalog
- Format
- Set ID
- Message ID

For GNU message catalogs these configuration data fields are relevant:

- Catalog
- Format

Set ID and Message ID are not relevant for GNU message catalog, since the GNU *gettext(3)* API uses the format string not only as default message text (as is the case with the POSIX *catgets(3)* API) but also as an index in the message catalog.

The format string may contain references to elements of the notification. At runtime the complete localized message text of a notification can be retrieved with the *saNtfLocalizedMessageGet* function of the Consumer API.

Field Name	Data Type	Description
NCI	SaNtfClassIdT	The Notification Class Identifier; index into this table (= unique key). The value {0, 0, 0} has a special meaning: If existing, it identifies the internationalization data which is used by default when no record is found for a particular NCI.
Catalog Type	enumeration: POSIX_TYPE GNU_TYPE	The type of the message catalog.
Catalog	string	Name of the localized message catalog (for POSIX) or domain (for GNU).
Format	string	The format string in the default language (english). It may contain references to elements in the notification. See below for a description of the syntax of the format string and the list of keywords which can be used for references. For GNU message catalogs this is also the message identifier passed to the gettext(3) function.
Set ID	int	The identifier of the message set (needed for POSIX catalogs, only)
Message ID	int	The identifier of the message within the set (needed for POSIX catalogs, only)

Currently, SA Forum does not define any internationalization data but provides the underlying mechanisms, only. All message catalogs and related configuration entries have to be provided by implementations.

The format string may contain keywords determining which notification parameter to insert at that place in the string. Keywords are inserted into the string with enclosing '${' and '}', for instance, "object ${notificationObject} created" to reference the

notification object. For list-type parameters, e.g., specific problems or object attributes, each list element is referenced via C style array syntax, for instance, “object ${notificationObject} created with xyz attribute value ${objectAttributes[0].attributeValue}” which references the first object attribute in the notification.

Elements of data type *SaNtfValueT* are inserted into the format string according to the field defining their data type (*SaNtfValueTypeT*). It is not supported to include elements of data type SA_NTF_VALUE_ARRAY or SA_NTF_VALUE_BINARY in the format text.

There are the following keywords (j represents an index into the respective array):

Keywords	**Notification Parameter**
eventType	Event Type
notificationObject	Notification Object
notifyingObject	Notifying Object
notificationClassIdentifier.vendorId notificationClassIdentifier.majorId notificationClassIdentifier.minorId	Notification Class Identifier
notificationIdentifier	Notification Identifier
correlatedNotifications[j]	Correlated Notifications
eventTime	Event Time
additionalText	Additional Text
additionalInformation[j].infoId additionalInformation[j].infoValue	Additional Information
probableCause	Probable cause
specificProblems[j].problemId specificProblems[j].problemClassId.vendorId specificProblems[j].problemClassId.majorId specificProblems[j].problemClassId.minorId specificProblems[j].problemValue	Specific problems
perceivedSeverity	Perceived severity
trendIndication	Trend indication

Keywords	Notification Parameter
thresholdInformation.thresholdId thresholdInformation.thresholdValue thresholdInformation.thresholdHysteresis thresholdInformation.observedValue thresholdInformation.armTime	Threshold information
monitoredAttributes[j].attributeId monitoredAttributes[j].attributeValue	Monitored attributes
proposedRepairActions[j].actionId proposedRepairActions[j]. actionValue	Proposed repair actions
sourceIndicator	Source indicator
changedStates[j].stateType changedStates[j].oldState changedStates[j].newState	Changed state attribute list
objectAttributes[j].attributeId objectAttributes[j].attributeValue	Attribute list (object creation/deletion)
changedAttributes[j].attributeId changedAttributes[j].oldAttributeValue changedAttributes[j].newAttributeValue	Changed attributes
securityAlarmCause	Security alarm cause
securityAlarmSeverity	Security alarm severity
securityAlarmDetector	Security alarm detector
serviceUser	Service user
serviceProvider	Service provider

Appendix A API Usage Examples

This section gives sample code for generating notifications using the API functions described in this document.

Producer Side (example function) – Object Create Delete Notification

```
/* Send a notification about the creation of an object of type AbcObject, which has two
attributes, one attribute of type integer and another attribute of type string. One additional
information element is used here to convey the current number of instances of this kind of
object. In this example, the object creation notification is correlated to a single previous
notification via the supplied correlatedId parameter. The notification identifier that is set by
saNtfNotificationSend() will be assigned to the supplied parameter ntfIdPtr.
This example uses a Notification Class Identifier with a vendorId 33333, majorId 999 and
minorId 1. The corresponding textual description of the situation is “Created
${notificationObject}, instance ${additionalInformation[0].infoValue} of AbcObject, with
integerAttr ${objectAttributes[0].attributeValue}”.
*/

SaAisErrorT sendAbcCreateNotification(
        SaNtfHandleT myNtfInstHandle,
        SaNameT *object,
        SaUint32T instCnt,
        SaInt32T integerAttrVal,
        SaStringT stringAttrVal,
        SaUint16T correlatedId,
        SaNtfIdentifierT *ntfIdPtr)
{
        SaNtfObjectCreateDeleteNotificationT myNotification;

        SaAisErrorT ret;

        SaStringT destPtr = NULL;
        SaStringT myAdditionalText = “My additional text”;

        /* identifier for meaning of first additional information item,
           in this case it is a counter for the current number of AbcObject instances */
        SaUint16T MY_INST_COUNT = 1;
```

```
ret = saNtfObjectCreateDeleteNotificationAllocate(
        myNtfInstHandle,              /* handle to Notification Service instance */
        &myNotification,
        1                             /* number of correlated notifications */,
        strlen(myAdditionalText) + 1  /* length of additional text */,
        1                             /* number of additional info items*/,
        2                             /* number of object attributes */,
        SA_NTF_ALLOC_SYSTEM_LIMIT /* use default allocation size */);

*(myNotification.notificationHeader.eventType) = SA_NTF_OBJECT_CREATION;

/* event time to be set automatically to current time by saNtfNotificationSend */
*(myNotification.notificationHeader.eventTime) = SA_TIME_UNKNOWN;

/* copy the given object name to notification storage */
myNotification.notificationHeader.notificationObject->length = object->length;
memcpy(myNotification.notificationHeader.notificationObject->value, object->value,
        object->length);

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myNotification.notificationHeader.notificationClassId->vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
myNotification.notificationHeader.notificationClassId->majorId = 999;
myNotification.notificationHeader.notificationClassId->minorId = 1;

/* who initiated this operation */
*(myNotification.sourceIndicator) = SA_NTF_OBJECT_OPERATION;

myNotification.notificationHeader.correlatedNotifications[0] = correlatedId;

/* set attributes */

/* object attributes have to be identified via the attributeId field, the list of
   possible values of attributeId is NCI and parameter specific, in this example
   the value is set to 1. */
myNotification.objectAttributes[0].attributeId = 1;
myNotification.objectAttributes[0].attributeType = SA_NTF_VALUE_INT32;
myNotification.objectAttributes[0].attributeValue.int32Val = integerAttrVal;
```

```
        /* object attributes have to be identified via the attributeId field, the list of possible
                values of attributeId is NCI and parameter specific, in this example the value
                is set to 2. */
        myNotification.objectAttributes[1].attributeId = 2;
        myNotification.objectAttributes[1].attributeType = SA_NTF_VALUE_STRING;

        ret = saNtfPtrValAllocate(
                myNotification.notificationHandle,
                strlen(stringAttrVal) + 1,
                (void**) &destPtr,
                &(myNotification.objectAttributes[1].attributeValue));

        if (ret != SA_AIS_OK) {
                fprintf(stderr, "could not allocate ptr value\n");
                saNtfNotificationFree(myNotification.notificationHandle);
                return ret;
        }
        strcpy(destPtr, stringAttrVal);

        /* set additional text*/
        strcpy(myNotification.notificationHeader.additionalText, myAdditionalText);

        /* set additional info item, in this case it is filled with the current number of AbcObject
           instances */
        myNotification.notificationHeader.additionalInfo[0].infoId = MY_INST_COUNT;

        myNotification.notificationHeader.additionalInfo[0].infoType =
                SA_NTF_VALUE_UINT32;
        myNotification.notificationHeader.additionalInfo[0].infoValue.uint32Val = instCnt;

        /* send notification, a unique notification identifier will be returned in ntfIdPtr */
        ret = saNtfNotificationSend(myNotification.notificationHandle);
        *ntfIdPtr = *(myNotification.notificationHeader.notificationId);

        ret = saNtfNotificationFree(myNotification.notificationHandle);

        return ret;
}
```

An example for calling the above function:

```
SaAisErrorT ret;
SaNtfIdentifierT ntfId;
SaNameT name;
SaNtfHandleT myNtfInstHandle;
…

/* inform about creation of first AbcObject instance with attribute values 33 and blue.
This notification is correlated to a previous notification with notification
identifier 100. */

ret = sendAbcCreateNotification(myNtfInstHandle, &name, 1, 33, “blue”, 100, &ntfId);
```

Producer Side (example function) – Attribute Change Notification

```
/* Send a notification about the modification of an object of type AbcObject, which has two
attributes, one attribute of type integer and another attribute of type string. In this example,
the attribute change notification is correlated to a single previous notification via the supplied
correlatedId parameter. The notification identifier that is set by saNtfNotificationSend() will
be assigned to the supplied parameter ntfIdPtr.
This example uses a Notification Class Identifier with a vendorId 33333, majorId 998 and
minorId 1. The corresponding textual description of the situation is
“Modified ${notificationObject}, instance of AbcObject, with new integerAttr
${changedAttributes[0].newAttributeValue}, stringAttr
${changedAttributes[1].newAttributeValue}”.
*/

SaAisErrorT sendAbcAttributeChangeNotification(
        SaNtfHandleT myNtfInstHandle,
        SaNameT *object,
        SaInt32T newIntegerAttrVal,
        SaInt32T oldIntegerAttrVal,
        SaStringT newStringAttrVal,
        SaStringT oldStringAttrVal,
        SaUint16T correlatedId,
        SaNtfIdentifierT *ntfIdPtr)
{
        SaNtfAttributeChangeNotificationT myNotification;

        SaAisErrorT ret;

        SaStringT destPtr = NULL;
        SaStringT myAdditionalText = “My additional text”;
        SaStringT myAdditionalInfo = “My second additional information item”;
        /* identifier for meaning of first additional information item */
        SaUint16T additionalInfoIdent1 = 2;
        /* identifier for meaning of second additional information item */
        SaUint16T additionalInfoIdent2 = 33;
```

```
ret = saNtfAttributeChangeNotificationAllocate(
        myNtfInstHandle,                /* handle to Notification Service instance */
        &myNotification,
        1                               /* number of correlated notifications */,
        strlen(myAdditionalText) + 1    /* length of additional text */,
        2                               /* number of additional info items*/,
        2                               /* number of object attributes */,
        SA_NTF_ALLOC_SYSTEM_LIMIT /* use default allocation size */);

*(myNotification.notificationHeader.eventType) = SA_NTF_ATTRIBUTE_CHANGED;

/* event time to be set automatically to current time by saNtfNotificationSend */
*(myNotification.notificationHeader.eventTime) = SA_TIME_UNKNOWN;

/* copy the given object name to notification storage */
myNotification.notificationHeader.notificationObject->length = object->length;
memcpy(myNotification.notificationHeader.notificationObject->value, object->value,
        object->length);

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myNotification.notificationHeader.notificationClassId->vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
myNotification.notificationHeader.notificationClassId->majorId = 998;
myNotification.notificationHeader.notificationClassId->minorId = 1;

/* who initiated this operation */
*(myNotification.sourceIndicator) = SA_NTF_OBJECT_OPERATION;

myNotification.notificationHeader.correlatedNotifications[0] = correlatedId;

/* set attributes */

/* reuse attributeId values from previous example */
myNotification.changedAttributes[0].attributeId = 1;
myNotification.changedAttributes[0].attributeType = SA_NTF_VALUE_INT32;
myNotification.changedAttributes[0].newAttributeValue.int32Val = newIntegerAttrVal;
myNotification.changedAttributes[0].oldAttributePresent = SA_TRUE;
myNotification.changedAttributes[0].oldAttributeValue.int32Val = oldIntegerAttrVal;
```

```
myNotification.changedAttributes[1].attributeId = 2;
myNotification.changedAttributes[1].attributeType = SA_NTF_VALUE_STRING;

ret = saNtfPtrValAllocate(
        myNotification.notificationHandle,
        strlen(newStringAttrVal) + 1,
        (void**) &destPtr,
        &(myNotification.changedAttributes[1].newAttributeValue));

if (ret != SA_AIS_OK) {
        fprintf(stderr, "could not allocate ptr value\n");
        saNtfNotificationFree(myNotification.notificationHandle);
        return ret;
}

strcpy(destPtr, newStringAttrVal);

myNotification.changedAttributes[1].oldAttributePresent = SA_TRUE;
ret = saNtfPtrValAllocate(
        myNotification.notificationHandle,
        strlen(oldStringAttrVal) + 1,
        (void**) &destPtr,
        &(myNotification.changedAttributes[1].oldAttributeValue));

if (ret != SA_AIS_OK) {
        fprintf(stderr, "could not allocate ptr value\n");
        saNtfNotificationFree(myNotification.notificationHandle);
        return ret;
}

strcpy(destPtr, oldStringAttrVal);

/* set additional text */
strcpy(myNotification.notificationHeader.additionalText, myAdditionalText);

/* set first additional info item */
myNotification.notificationHeader.additionalInfo[0].infoId = additionalInfoIdent1;
myNotification.notificationHeader.additionalInfo[0].infoType =
        SA_NTF_VALUE_INT32;
myNotification.notificationHeader.additionalInfo[0].infoValue.int32Val = 100;
```

```
        /* set second additional info item */
        myNotification.notificationHeader.additionalInfo[1].infoId = additionalInfoIdent2;
        myNotification.notificationHeader.additionalInfo[1].infoType =
                SA_NTF_VALUE_STRING;

        ret = saNtfPtrValAllocate(
                myNotification.notificationHandle,
                strlen(myAdditionalInfo) + 1,
                (void**) &destPtr,
                &(myNotification.notificationHeader.additionalInfo[1].infoValue));

        if (ret != SA_AIS_OK) {
                fprintf(stderr, "could not allocate ptr value\n");
                saNtfNotificationFree(myNotification.notificationHandle);
                return ret;
        }

        strcpy(destPtr, myAdditionalInfo);

        ret = saNtfNotificationSend(myNotification.notificationHandle);
        *ntfIdPtr = *(myNotification.notificationHeader.notificationId);

        ret = saNtfNotificationFree(myNotification.notificationHandle);

        return ret;
}
```

An example for calling the above function:

```
SaAisErrorT ret;
SaNtfIdentifierT ntfId;
SaNameT name;
SaNtfHandleT myNtfInstHandle;
…

/* Inform about changes in an AbcObject instance with attribute changes from
33 to 42 and from blue to red. This notification is correlated to a previous notification
 with notification identifier 101. */

ret = sendAbcAttributeChangeNotification(myNtfInstHandle, &name, 42, 33, "red",
                                         "blue", 101, &ntfId);
```

Producer Side (example function) – State Change Notification

```
/* Send a notification about the state changes of an object of type AbcObject, which has two
state attributes: operational state and usage state. In this example, the state change
notification is correlated to a single previous notification via the supplied correlatedId
parameter. The notification identifier that is set by saNtfNotificationSend() will be assigned to
the supplied parameter ntfIdPtr.
This example uses a Notification Class Identifier with a vendorId 33333, majorId 997 and
minorId 1. The corresponding textual description of the situation is
"${notificationObject} with new operational state ${changedStates[0].newState} and new
usage state ${changedStates[1].newState}".
*/

/* application-specific definition of element ID for operational state and usage state */
#define MY_APP_OPER_STATE 1
#define MY_APP_USAGE_STATE 2

SaAisErrorT sendAbcStateChangeNotification(
        SaNtfHandleT myNtfInstHandle,
        SaNameT *object,
        SaUint32T newOpState,
        SaUint32T oldOpState,
        SaUint32T newUsgState,
        SaUint32T oldUsgState,
        SaUint16T correlatedId,
        SaNtfIdentifierT *ntfIdPtr)
{
        SaNtfStateChangeNotificationT myNotification;

        SaAisErrorT ret;

        SaStringT destPtr = NULL;
        SaStringT myAdditionalText = "My additional text";
        SaStringT myAdditionalInfo = "My second additional information item";
        /* identifier for meaning of first additional information item */
        SaUint16T additionalInfoIdent1 = 2;
        /* identifier for meaning of second additional information item */
        SaUint16T additionalInfoIdent2 = 33;
```

```
ret = saNtfStateChangeNotificationAllocate(
        myNtfInstHandle,              /* handle to Notification Service instance */
        &myNotification,
        1                             /* number of correlated notifications */,
        strlen(myAdditionalText) + 1  /* length of additional text */,
        2                             /* number of additional info items*/,
        2                             /* number of changed object states */,
        SA_NTF_ALLOC_SYSTEM_LIMIT /* use default allocation size */);


*(myNotification.notificationHeader.eventType) =
        SA_NTF_OBJECT_STATE_CHANGE;

/* event time to be set automatically to current time by saNtfNotificationSend */
*(myNotification.notificationHeader.eventTime) = SA_TIME_UNKNOWN;

/* copy the given object name to notification storage */
myNotification.notificationHeader.notificationObject->length = object->length;
memcpy(myNotification.notificationHeader.notificationObject->value, object->value,
        object->length);

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myNotification.notificationHeader.notificationClassId->vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
myNotification.notificationHeader.notificationClassId->majorId = 997;
myNotification.notificationHeader.notificationClassId->minorId = 1;

/* who initiated this operation */
*(myNotification.sourceIndicator) = SA_NTF_OBJECT_OPERATION;

myNotification.notificationHeader.correlatedNotifications[0] = correlatedId;

/* set operational state */
myNotification.changedStates[0].stateId = MY_APP_OPER_STATE;
myNotification.changedStates[0].oldStatePresent = SA_TRUE;
myNotification.changedStates[0].oldState = oldOpState;
myNotification.changedStates[0].newState = newOpState;
```

```
        /* set usage state */
        myNotification.changedStates[1].stateId = MY_APP_USAGE_STATE;
        myNotification.changedStates[1].oldStatePresent = SA_TRUE;
        myNotification.changedStates[1].oldState = oldUsgState;
        myNotification.changedStates[1].newState = newUsgState;

        /* set additional text */
        strcpy(myNotification.notificationHeader.additionalText, myAdditionalText);

        /* set first additional info item */
        myNotification.notificationHeader.additionalInfo[0].infoId = additionalInfoIdent1;
        myNotification.notificationHeader.additionalInfo[0].infoType =
                SA_NTF_VALUE_UINT8;
        myNotification.notificationHeader.additionalInfo[0].infoValue.uint8Val = 42;

        /* set second additional info item */
        myNotification.notificationHeader.additionalInfo[1].infoId = additionalInfoIdent2;
        myNotification.notificationHeader.additionalInfo[1].infoType =
                SA_NTF_VALUE_STRING;

        ret = saNtfPtrValAllocate(
                myNotification.notificationHandle,
                strlen(myAdditionalInfo) + 1,
                (void**) &destPtr,
                &(myNotification.notificationHeader.additionalInfo[1].infoValue));

        if (ret != SA_AIS_OK) {
                fprintf(stderr, "could not allocate ptr value\n");
                saNtfNotificationFree(myNotification.notificationHandle);
                return ret;
        }

        strcpy(destPtr, myAdditionalInfo);

        ret = saNtfNotificationSend(myNotification.notificationHandle);
        *ntfIdPtr = *(myNotification.notificationHeader.notificationId);

        ret = saNtfNotificationFree(myNotification.notificationHandle);

        return ret;
}
```

An example for calling the above function:

```
SaAisErrorT ret;
SaNtfIdentifierT ntfId;
SaNameT name;
SaNtfHandleT myNtfInstHandle;
…

/* Inform about state changes in an AbcObject instance with state changes from
SA_NTF_DISABLED to SA_NTF_ENABLED and from SA_NTF_IDLE to
SA_NTF_ACTIVE. This notification is correlated to a previous notification with
notification identifier 102. */

ret = sendAbcStateChangeNotification(
        myNtfInstHandle
        &name,
        SA_NTF_ENABLED,
        SA_NTF_DISABLED,
        SA_NTF_ACTIVE,
        SA_NTF_IDLE,
        102,
        &ntfId);
```

Producer Side (example function) – Alarm Notification

```
/* Send an alarm notification about an object of type AbcObject that has communication
problems caused by reduced bandwidth. In this example, the alarm notification is correlated
to a single previous notification via the supplied correlatedId parameter. This example
shows two specific problems and two repair actions.Notification parameters are partly
supplied as arguments to this example function and partly hard-coded.The notification
identifier that is set by saNtfNotificationSend() will be assigned to the supplied parameter
ntfIdPtr.
This example uses a Notification Class Identifier with a vendorId 33333, majorId 996 and
minorId 1. The corresponding textual description of the situation is
“Communication problem: reduced bandwidth on connections
${specificProblems[0].problemValue} (${additionalInformation[0].infoValue} %) and
${specificProblems[1].problemValue} ((${additionalInformation[1].infoValue} %)”.
*/

SaAisErrorT sendAbcAlarmNotification(
        SaNtfHandleT myNtfInstHandle,
        SaNameT *object,
        SaInt32T specificProblem1,
        SaUint16T perc1,
        SaInt32T specificProblem2,
        SaUint16T perc2,
        SaUint16T repair1,
        SaUint16T repair2,
        SaUint16T correlatedId,
        SaNtfIdentifierT *ntfIdPtr)
{
        SaNtfAlarmNotificationT myNotification;

        SaAisErrorT ret;

        SaStringT destPtr = NULL;
        SaStringT myAdditionalText = “My additional text”;
        SaStringT myAttribute2 = “My Attribute”;
        SaStringT myRepairArguments1 = “connection1”;
        SaStringT myRepairArguments2 = “connection2”;

        SaNtfElementIdT MY_CONNECTION = 1; /* my application specific problem id */
        SaNtfElementIdT MY_PERCENTAGE = 1; /* my application specific additional
                information id */
```

```
ret = saNtfAlarmNotificationAllocate(
        myNtfInstHandle,                /* handle to Notification Service instance */
        &myNotification,
        1                               /* number of correlated notifications */,
        strlen(myAdditionalText) + 1    /* length of additional text */,
        2                               /* number of additional info items*/,
        2                               /* number of specific problems */,
        2                               /* number of monitored attributes */,
        2                               /* number of proposed repair actions */,
        SA_NTF_ALLOC_SYSTEM_LIMIT /* use default allocation size */);

*(myNotification.notificationHeader.eventType) =
        SA_NTF_ALARM_COMMUNICATION;

/* event time to be set automatically to current time by saNtfNotificationSend */
*(myNotification.notificationHeader.eventTime) = SA_TIME_UNKNOWN;

/* copy the given object name to notification storage */
myNotification.notificationHeader.notificationObject->length = object->length;
memcpy(myNotification.notificationHeader.notificationObject->value, object->value,
        object->length);

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myNotification.notificationHeader.notificationClassId->vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
myNotification.notificationHeader.notificationClassId->majorId = 996;
myNotification.notificationHeader.notificationClassId->minorId = 1;

/* determine perceived severity */
*(myNotification.perceivedSeverity) = SA_NTF_SEVERITY_MAJOR;

/* determine trend indication */
*(myNotification.trend) = SA_NTF_TREND_NO_CHANGE;

myNotification.notificationHeader.correlatedNotifications[0] = correlatedId;

/* set probable cause*/
*(myNotification.probableCause) = SA_NTF_BANDWIDTH_REDUCED;
```

```
/* set first specific problem */
myNotification.specificProblems[0].problemId = MY_CONNECTION;
/* no reference to other NCI, set problemClassId values to 0 */
myNotification.specificProblems[0].problemClassId.vendorId = 0;
myNotification.specificProblems[0].problemClassId.majorId = 0;
myNotification.specificProblems[0].problemClassId.minorId = 0;
myNotification.specificProblems[0].problemType = SA_NTF_VALUE_INT32;
myNotification.specificProblems[0].problemValue.int32Val = specificProblem1;

/* set second specific problem */
myNotification.specificProblems[1].problemId = MY_CONNECTION;
/* no reference to other NCI, set problemClassId values to 0 */
myNotification.specificProblems[1].problemClassId.vendorId = 0;
myNotification.specificProblems[1].problemClassId.majorId = 0;
myNotification.specificProblems[1].problemClassId.minorId = 0;

myNotification.specificProblems[1].problemType= SA_NTF_VALUE_INT32;
myNotification.specificProblems[1].problemValue.int32Val = specificProblem2;

/* set first proposed repair action */
myNotification.proposedRepairActions[0].actionId = repair1;

myNotification.proposedRepairActions[0].actionValueType =
        SA_NTF_VALUE_STRING;

ret = saNtfPtrValAllocate(
        myNotification.notificationHandle,
        strlen(myRepairArguments1) + 1,
        (void**) &destPtr,
        &(myNotification.proposedRepairActions[0].actionValue));

if (ret != SA_AIS_OK) {
        fprintf(stderr, “could not allocate ptr value\n”);
        saNtfNotificationFree(myNotification.notificationHandle);
        return ret;
}

strcpy(destPtr, myRepairArguments1);
```

```
/* set second proposed repair action */
myNotification.proposedRepairActions[1].actionId = repair2;
myNotification.proposedRepairActions[1].actionValueType =
        SA_NTF_VALUE_STRING;

ret = saNtfPtrValAllocate(
        myNotification.notificationHandle,
        strlen(myRepairArguments2) + 1,
        (void**) &destPtr,
        &(myNotification.proposedRepairActions[1].actionValue));

if (ret != SA_AIS_OK) {
        fprintf(stderr, "could not allocate ptr value\n");
        saNtfNotificationFree(myNotification.notificationHandle);
        return ret;
}

strcpy(destPtr, myRepairArguments2);

/* set first monitored attribute; object attributes have to be identified via the attributeId
        field, the list of possible values of attributeId is NCI specific, in this example the
        value is set to 1. */
myNotification.monitoredAttributes[0].attributeId = 1;
myNotification.monitoredAttributes[0].attributeType = SA_NTF_VALUE_INT32;
myNotification.monitoredAttributes[0].attributeValue.int32Val = 100;

/* set second monitored attribute; object attributes have to be identified via the
        attributeId field, the list of possible values of attributeId is object and NCI
        specific, in this example the value is set to 2. */
myNotification.monitoredAttributes[1].attributeId = 2;
myNotification.monitoredAttributes[1].attributeType = SA_NTF_VALUE_STRING;

ret = saNtfPtrValAllocate(
        myNotification.notificationHandle,
        strlen(myAttribute2) + 1,
        (void**) &destPtr,
        &(myNotification.monitoredAttributes[1].attributeValue));
```

```
        if (ret != SA_AIS_OK) {
                fprintf(stderr, "could not allocate ptr value\n");
                saNtfNotificationFree(myNotification.notificationHandle);
                return ret;
        }

        strcpy(destPtr, myAttribute2);

        /* set additional text and additional info */

        strcpy(myNotification.notificationHeader.additionalText, myAdditionalText);

        /* set first additional info item, in this case it contains a percentage value */
        myNotification.notificationHeader.additionalInfo[0].infoId =
                MY_PERCENTAGE;
        myNotification.notificationHeader.additionalInfo[0].infoType =
                SA_NTF_VALUE_UINT16;
        myNotification.notificationHeader.additionalInfo[0].infoValue.uint16Val = perc1;

        /* set second additional info item, in this case it contains a percentage value */
        myNotification.notificationHeader.additionalInfo[1].infoId =
                MY_PERCENTAGE;
        myNotification.notificationHeader.additionalInfo[1].infoType =
                SA_NTF_VALUE_UINT16;
        myNotification.notificationHeader.additionalInfo[1].infoValue.uint16Val = perc2;

        ret = saNtfNotificationSend(myNotification.notificationHandle);
        *ntfIdPtr = *(myNotification.notificationHeader.notificationId);

        ret = saNtfNotificationFree(myNotification.notificationHandle);

        return ret;
}
```

An example for calling the above function:

```
SaAisErrorT ret;
SaNtfIdentifierT ntfId;
SaNameT name;
SaNtfHandleT myNtfInstHandle;
…

/* Inform about communication problems of an AbcObject instance with a loss of 5 %
on its connection identified by 1034 and 3 % on connection 1035. Repair actions are
given by 1 and 2. This notification is correlated to a previous notification with
notification identifier 111. */

ret = sendAbcAlarmNotification(
        myNtfInstHandle,
        &name,
        1034,
        5,
        1035,
        3,
        1,
        2,
        111,
        &ntfId);
```

Producer Side (example function) – Security Alarm Notification

```
/* Send a security alarm notification about an authentication failure for accessing an object
of type AbcObject. In this example, the notification is correlated to a single previous
notification via the supplied correlatedId parameter.The notification identifier that is set by
saNtfNotificationSend() will be assigned to the supplied parameter ntfIdPtr.
This example uses a Notification Class Identifier with a vendorId 33333, majorId 995 and
minorId 1. The corresponding textual description of the situation is
“Service provider ${serviceProvider}: authentication failure of service user ${serviceUser}”.
*/

SaAisErrorT sendAbcSecurityAlarmNotification(
        SaNtfHandleT myNtfInstHandle,
        SaNameT *object,
        SaStringT serviceUser,
        SaStringT serviceProvider,
        SaStringT alarmDetector,
        SaUint16T correlatedId,
        SaNtfIdentifierT *ntfIdPtr)
{
        SaNtfSecurityAlarmNotificationT myNotification;

        SaAisErrorT ret;

        SaStringT destPtr = NULL;

        ret = saNtfSecurityAlarmNotificationAllocate(
                myNtfInstHandle,            /* handle to Notification Service instance */
                &myNotification,
                1                           /* number of correlated notifications */,
                0                           /* length of additional text */,
                0                           /* number of additional info items*/,
                SA_NTF_ALLOC_SYSTEM_LIMIT /* use default allocation size */);


        *(myNotification.notificationHeader.eventType) =
                SA_NTF_OPERATION_VIOLATION;

        /* event time to be set automatically to current time by saNtfNotificationSend */
        *(myNotification.notificationHeader.eventTime) = SA_TIME_UNKNOWN;
```

```
/* copy the given object name to notification storage */
myNotification.notificationHeader.notificationObject->length = object->length;
memcpy(myNotification.notificationHeader.notificationObject->value, object->value,
        object->length);

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myNotification.notificationHeader.notificationClassId->vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
myNotification.notificationHeader.notificationClassId->majorId = 995;
myNotification.notificationHeader.notificationClassId->minorId = 1;

/* set severity */
*(myNotification.severity) = SA_NTF_SEVERITY_MAJOR;

myNotification.notificationHeader.correlatedNotifications[0] = correlatedId;

/* set probable cause */
*(myNotification.probableCause) = SA_NTF_AUTHENTICATION_FAILURE;

/* set service user; a string is used here */
myNotification.serviceUser->valueType = SA_NTF_VALUE_STRING;
ret = saNtfPtrValAllocate(myNotification.notificationHandle, strlen(serviceUser) + 1,
                          (void**) &destPtr, &(myNotification.serviceUser->value));
if (ret != SA_AIS_OK) {
        fprintf(stderr, “could not allocate ptr value\n”);
        saNtfNotificationFree(myNotification.notificationHandle);
        return ret;
}
strcpy(destPtr, serviceUser);

/* set service provider; a string is used here */
myNotification.serviceProvider->valueType = SA_NTF_VALUE_STRING;
ret = saNtfPtrValAllocate(myNotification.notificationHandle,
                          strlen(serviceProvider) + 1,
                          (void**) &destPtr,
                          &(myNotification.serviceProvider->value));
if (ret != SA_AIS_OK) {
        fprintf(stderr, “could not allocate ptr value\n”);
        saNtfNotificationFree(myNotification.notificationHandle);
```

```
        return ret;
    }
    strcpy(destPtr, serviceProvider);

    /* set alarm detector; a string is used here */
    myNotification.securityAlarmDetector->valueType = SA_NTF_VALUE_STRING;
    ret = saNtfPtrValAllocate(myNotification.notificationHandle,
                              strlen(alarmDetector) + 1, (void**) &destPtr,
                              &(myNotification.securityAlarmDetector->value));

    if (ret != SA_AIS_OK) {
        fprintf(stderr, "could not allocate ptr value\n");
        saNtfNotificationFree(myNotification.notificationHandle);
        return ret;
    }
    strcpy(destPtr, alarmDetector);

    /* No additional text and no additional info */

    ret = saNtfNotificationSend(myNotification.notificationHandle);
    *ntfIdPtr = *(myNotification.notificationHeader.notificationId);

    ret = saNtfNotificationFree(myNotification.notificationHandle);

    return ret;

}
```

An example for calling the above function:

```
SaAisErrorT ret;
SaNtfIdentifierT ntfId;
SaNameT name;
SaNtfHandleT myNtfInstHandle;
…

/* Inform about an authentication error for accessing an AbcObject instance.
This notification is correlated to a previous notification with notification i
identifier 120. */

ret = sendAbcSecurityAlarmNotification(
        myNtfInstHandle,
        &name,
        “My Service User”,
        “My Service Provider”,
        “My Alarm Detector”,
        120,
        &ntfId);
```

Consumer Side (example function) – Subscribe for Notifications

```
/* subscribe for all those kinds of notifications that are generated in the producer API
examples using the notification class identifiers as filter criteria.
*/
SaAisErrorT subscribeForAbcNotifications(SaNtfHandleT myNtfInstHandle)
{
        SaAisErrorT ret;

        SaNtfObjectCreateDeleteNotificationFilterT myOCDFilter;
        SaNtfAttributeChangeNotificationFilterT myAVCFilter;
        SaNtfStateChangeNotificationFilterT mySCFilter;
        SaNtfAlarmNotificationFilterT myAFilter;
        SaNtfSecurityAlarmNotificationFilterT mySAFilter;

        SaNtfNotificationTypeFilterHandlesT abcNotificationFilterHandles;


        ret = saNtfObjectCreateDeleteNotificationFilterAllocate(
                myNtfInstHandle,            /* handle to Notification Service instance */
                &myOCDFilter,               /* put filter here */
                0                           /* number of event types */,
                0                           /* number of notification objects */,
                0                           /* number of notifying objects */,
                1                           /* number of notification class ids */,
                0                           /* number of source indicators */);
        if (ret != SA_AIS_OK)
        {
                fprintf(stderr, "could not allocate notification filter \n");
                return ret;
        }

        /* set Notification Class Identifier */

        /* vendor id 33333 is not an existing SNMP enterprise number - just an example */
        myOCDFilter.notificationFilterHeader.notificationClassIds[0].vendorId = 33333;

        /* sub id of this notification class within "name space" of vendor ID */
        myOCDFilter.notificationFilterHeader.notificationClassIds[0].majorId = 999;
        myOCDFilter.notificationFilterHeader.notificationClassIds[0].minorId = 1;
```

```
ret = saNtfAttributeChangeNotificationFilterAllocate(
        myNtfInstHandle,                /* handle to Notification Service instance */
        &myAVCFilter,                   /* put filter here */
        0                               /* number of event types */,
        0                               /* number of notification objects */,
        0                               /* number of notifying objects */,
        1                               /* number of notification class ids */,
        0                               /* number of source indicators */);

if (ret != SA_AIS_OK)
{
        fprintf(stderr, “could not allocate notification filter \n”);
        saNtfNotificationFilterFree(myOCDFilter.notificationFilterHandle);
        return ret;
}

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myAVCFilter.notificationFilterHeader.notificationClassIds[0].vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
myAVCFilter.notificationFilterHeader.notificationClassIds[0].majorId = 998;
myAVCFilter.notificationFilterHeader.notificationClassIds[0].minorId = 1;

ret = saNtfStateChangeNotificationFilterAllocate(
        myNtfInstHandle,                /* handle to Notification Service instance */
        &mySCFilter,                    /* put filter here */
        0                               /* number of event types */,
        0                               /* number of notification objects */,
        0                               /* number of notifying objects */,
        1                               /* number of notification class ids */,
        0                               /* number of source indicators */,
        0                               /* number of changed states */);
```

```
if (ret != SA_AIS_OK)
{
        fprintf(stderr, "could not allocate notification filter \n");
        saNtfNotificationFilterFree(myOCDFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myAVCFilter.notificationFilterHandle);
        return ret;
}

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
mySCFilter.notificationFilterHeader.notificationClassIds[0].vendorId = 33333;

/* sub id of this notification class within "name space" of vendor ID */
mySCFilter.notificationFilterHeader.notificationClassIds[0].majorId = 997;
mySCFilter.notificationFilterHeader.notificationClassIds[0].minorId = 1;


ret = saNtfAlarmNotificationFilterAllocate(
        myNtfInstHandle,            /* handle to Notification Service instance */
        &myAFilter,                 /* put filter here */
        0                           /* number of event types */,
        0                           /* number of notification objects */,
        0                           /* number of notifying objects */,
        1                           /* number of notification class ids */,
        0                           /* number of probable causes */,
        0                           /* number of perceived severities */,
        0                           /* number of trend indications */);

if (ret != SA_AIS_OK)
{
        fprintf(stderr, "could not allocate notification filter \n");
        saNtfNotificationFilterFree(myOCDFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myAVCFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(mySCFilter.notificationFilterHandle);
        return ret;
}

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
myAFilter.notificationFilterHeader.notificationClassIds[0].vendorId = 33333;
```

```
/* sub id of this notification class within “name space” of vendor ID */
myAFilter.notificationFilterHeader.notificationClassIds[0].majorId = 996;
myAFilter.notificationFilterHeader.notificationClassIds[0].minorId = 1;


ret = saNtfSecurityAlarmNotificationFilterAllocate(
        myNtfInstHandle,            /* handle to Notification Service instance */
        &mySAFilter,                /* put filter here */
        0                           /* number of event types */,
        0                           /* number of notification objects */,
        0                           /* number of notifying objects */,
        1                           /* number of notification class ids */,
        0                           /* number of probable causes */,
        0                           /* number of severities */,
        0                           /* number of security alarm detectors */,
        0                           /* number of service users */,
        0                           /* number of service providers */);

if (ret != SA_AIS_OK)
{
        fprintf(stderr, “could not allocate notification filter \n”);
        saNtfNotificationFilterFree(myOCDFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myAVCFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(mySCFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myAFilter.notificationFilterHandle);
        return ret;
}

/* set Notification Class Identifier */

/* vendor id 33333 is not an existing SNMP enterprise number - just an example */
mySAFilter.notificationFilterHeader.notificationClassIds[0].vendorId = 33333;

/* sub id of this notification class within “name space” of vendor ID */
mySAFilter.notificationFilterHeader.notificationClassIds[0].majorId = 995;
mySAFilter.notificationFilterHeader.notificationClassIds[0].minorId = 1;
```

```
        abcNotificationFilterHandles.objectCreateDeleteFilterHandle =
                myOCDFilter.notificationFilterHandle;
        abcNotificationFilterHandles.attributeChangeFilterHandle =
                myAVCFilter.notificationFilterHandle;
        abcNotificationFilterHandles.stateChangeFilterHandle =
                mySCFilter.notificationFilterHandle;
        abcNotificationFilterHandles.alarmFilterHandle =
                myAFilter.notificationFilterHandle;
        abcNotificationFilterHandles.securityAlarmFilterHandle =
                mySAFilter.notificationFilterHandle;

        ret = saNtfNotificationSubscribe(&abcNotificationFilterHandles, 1);

        saNtfNotificationFilterFree(myOCDFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myACFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(mySCFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myAFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(mySAFilter.notificationFilterHandle);

        return ret;

}

/* define common callback for all notification types */

void myNotificationCallback(
        SaNtfSubscriptionIdT subscriptionId,
        const SaNtfNotificationsT *notification
)
{
        SaNtfNotificationHeaderT *ntfHeader;
        SaNtfNotificationHandleT ntfHandle;
        void myNtfGenericHandler(
                  SaNtfNotificationHandleT notificationHandle,
                  SaNtfNotificationHeaderT * header);

        switch(notification->notificationType)
        {
        case SA_NTF_TYPE_OBJECT_CREATE_DELETE:
                ntfHeader =
                 &(notification->notification.objectCreateDeleteNotification.notificationHeader);
```

```
        ntfHandle =
          notification->notification.objectCreateDeleteNotification.notificationHandle;
        break;
case SA_NTF_TYPE_ATTRIBUTE_CHANGE:
        ntfHeader =
            &(notification->notification.attributeChangeNotification.notificationHeader);
        ntfHandle =
             notification->notification.attributeChangeNotification.notificationHandle;
        break;
case SA_NTF_TYPE_STATE_CHANGE:
        ntfHeader =
                &(notification->notification.stateChangeNotification.notificationHeader);
        ntfHandle =
                notification->notification.stateChangeNotification.notificationHandle;
        break;
case SA_NTF_TYPE_ALARM:
        ntfHeader = &(notification->notification.alarmNotification.notificationHeader);
        ntfHandle = notification->notification.alarmNotification.notificationHandle;
        break;
case SA_NTF_TYPE_SECURITY_ALARM:
        ntfHeader =
              &(notification->notification.securityAlarmNotification.notificationHeader);
        ntfHandle =
                notification->notification.securityAlarmNotification.notificationHandle;
        break;
}
/* first do some generic notification handling */
myNtfGenericHandler(ntfHandle, ntfHeader);

/* then do some handling specific for the notification type */
/* ... */
switch(notification->notificationType)
{
case SA_NTF_TYPE_OBJECT_CREATE_DELETE:
        /* ... */
        break;
case SA_NTF_TYPE_ATTRIBUTE_CHANGE:
        /* ... */
        break;
case SA_NTF_TYPE_STATE_CHANGE:
        /* ... */
        break;
```

```
        case SA_NTF_TYPE_ALARM:
                /* ... */
                break;
        case SA_NTF_TYPE_SECURITY_ALARM:
                /* ... */
                break;
        }

        /* free resources */
        saNtfNotificationFree(ntfHandle);
}


/* some simple generic handling for all kinds of notifications,
 * simply print their message text to stdout together with the notification time stamp. */
```

```
void myNtfGenericHandler(
        SaNtfNotificationHandleT notificationHandle,
        SaNtfNotificationHeaderT * header
)
{
        SaAisErrorT rc;
        SaStringT message = (SaStringT) NULL;
        char time_str[24];
        SaTimeT ntfTime = (SaTimeT)0;

        rc = saNtfLocalizedMessageGet(notificationHandle, &message);
        if (rc != SA_AIS_OK)
        {
                fprintf(stderr, “Cannot get localized message text, error %d\n, rc);
                return;
        }
        ntfTime = *(header->eventTime);
        ntfTime /= SA_TIME_ONE_SECOND;

        /* print message together with some info from notification, e.g., time */
        strftime(time_str, sizeof(time_str), "%d-%m-%Y %T", localtime((time_t *) &ntfTime));
        printf(“%s %s\n”, time_str, message);

}
```

An example for using the above functions:

```
SaAisErrorT ret;
SaVersionT version;
SaNtfHandleT ntfHandle,
…

/* set up callback */

ntfCallbacks.saNtfNotificationCallback = myNotificationCallback;

/* then initialize library instance */
ret = saNtfInitialize(&ntfHandle, &ntfCallbacks, &version);
if (ret != SA_AIS_OK)
{
        /* could not initialize the notification service library */
        exit (1);
}

/* subscribe for notifications */

ret = subscribeForAbcNotifications(ntfHandle);

…
```

Consumer Side (example function) – Read Logged Notifications

```
/* In this example, it is assumed that an application has subscribed for certain alarm
notifications, but due to a short application down time (e.g., due to a crash) or fail over, it
may have lost some notifications. Reading logged notifications will cover the gap. The time
stamp of the last received notification is used as a starting point. */

SaAisErrorT readMissedAbcNotifications(
        SaNtfHandleT myNtfInstHandle,
        SaNtfIdentifierT lastReceivedNotificationId
)
{
        SaAisErrorT ret;
        SaStringT destPtr = NULL;
        SaNtfNotificationsT returnedNotification;
        SaNtfNotificationTypeFilterHandlesT notificationFilterHandles;
        SaNtfSearchCriteriaT criteria;
        SaNtfAlarmNotificationFilterT myAFilter;
        SaNtfReadHandleT readHandle;

        ret = saNtfAlarmNotificationFilterAllocate(
                myNtfInstHandle,        /* handle to Notification Service instance */
                &myAFilter,                     /* put filter here */
                0                               /* number of event types */,
                0                               /* number of notification objects */,
                0                               /* number of notifying objects */,
                1                               /* number of notification class ids */,
                0                               /* number of probable causes */,
                0                               /* number of perceived severities */,
                0                               /* number of trend indications */,
                0                               /* number of source indicators */);

        if (ret != SA_AIS_OK)
        {
        fprintf(stderr, “could not allocate notification filter \n”);
        return ret;
        }

        /* set Notification Class Identifier */

        /* vendor id 33333 is not an existing SNMP enterprise number - just an example */
        myAFilter.notificationFilterHeader.notificationClassIds[0].vendorId = 33333;
```

```
/* sub id of this notification class within "name space" of vendor ID */
myAFilter.notificationHeader.notificationClassIds[0].majorId = 990;
myAFilter.notificationHeader.notificationClassIds[0].minorId = 0x12;


notificationFilterHandles.alarmFilterHandle =
        myAFilter.notificationFilterHandle;
notificationFilterHandles.objectCreateDeleteFilterHandle =
        SA_NTF_FILTER_HANDLE_NULL;
notificationFilterHandles.attributeChangeFilterHandle =
        SA_NTF_FILTER_HANDLE_NULL;
notificationFilterHandles.stateChangeFilterHandle =
        SA_NTF_FILTER_HANDLE_NULL;
notificationFilterHandles.securityAlarmFilterHandle =
        SA_NTF_FILTER_HANDLE_NULL;

/* initial search criteria is the last notification ID that was received before the
        application down time */
criteria.searchMode = SA_NTF_NOTIFICATION_ID;
criteria.notificationId = lastReceivedNotificationId;
ret = saNtfNotificationReadInitialize(criteria,
                                      &notificationFilterHandles,
                                      &readHandle);
if (ret != SA_AIS_OK)
{
        fprintf(stderr, "could not initialize read sequence with last received
                notification id\n");
        return ret;
}
/* filters no longer needed - free them */
saNtfNotificationFilterFree(myAFilter.notificationFilterHandle);

/* read as many matching notifications as exist for the time period between the
   last received one and now */
for (; (ret = saNtfNotificationReadNext(readHandle,
                                      SA_NTF_SEARCH_YOUNGER,
                                       &returnedNotification)) == SA_AIS_OK; )
{
```

```
        SaInt8 *dataPtr;
        SaUint16T dataSize;
        SaNtfAlarmNotificationT *returnedANtf;

        returnedANtf = &returnedNotification.alarmNotification;

        …
        /* handle this notification, e.g., check whether the application has not yet
           received it previously */
        …

        /* get first proposed repair action in alarm notification */
        if (returnedANtf->numProposedRepairActions > 0 &&
            returnedANtf->proposedRepairActions[0].actionValueType ==
                SA_NTF_VALUE_STRING)
        {
                ret = saNtfPtrValGet(
                        returnedANtf->notificationHandle,
                        &(returnedANtf->proposedRepairActions[0].actionValue),
                        &dataPtr,
                        &dataSize);
                … /* do something with the proposed repair action value pointed to by
                        dataPtr */
        }
        saNtfNotificationFree(returnedANtf->notificationHandle);
    }

    /* finalize reading */
    saNtfNotificationReadFinalize(readHandle);

    return ret;
}
```