Service Availability™ Forum Application Interface Specification

Notification Service SAI-AIS-NTF-A.03.01

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

The Service Availability™ Forum Application Interface Specification (the "Package") found at the URL http://www.saforum.org is generally made available by the Service Availability Forum (the "Copyright Holder") 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 Package are covered by the Artistic License 2.0 of the Perl Foundation, which is reproduced here.

The Artistic License 2.0

Copyright (c) 2000-2006, The Perl Foundation.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed.

The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.

"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

Permission for Use and Modification Without Distribution

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

Permissions for Redistribution of the Standard Version

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

Distribution of Modified Versions of the Package as Source

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution.

If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

General Provisions

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty:

THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents Notification Service

1 Document Introduction 11

1.1 Document Purpose 11
1.2 AIS Documents Organization 11
1.3 History 11
1.3.1 New Topics 11
1.3.2 Clarifications 14
1.3.3 Deleted Topics 14
1.3.4 Other Changes 15
1.3.5 Superseded and Superseding Functions 15
1.3.6 Changes in Return Values of API and Administrative Functions 17
1.4 References 17
1.5 How to Provide Feedback on the Specification 18
1.6 How to Join the Service Availability™ Forum 18
1.7 Additional Information 19
1.7.1 Member Companies 19
1.7.2 Press Materials 19

2 Overview 21

2.1 Users of the Notification Service Library 23
2.1.1 Producer 23
2.1.2 Consumer 23
2.1.2.1 Subscriber 23
2.1.2.2 Reader 23
2.2 Notification Service 23
2.2.1 Notification Service Library 23
2.2.2 Notification Server 23
2.2.3 Transport Service 24
2.3 Log Service 24

3 Notification Service API 25

3.1 Notifications 25
3.2 Notification Filters 25
3.3 Notification Types 25
3.3.1 Alarm Notification 25
3.3.2 State Change Notification 26
3.3.3 Object Create/Delete and Attribute Change Notifications 26
3.3.4 Security Alarm Notification 26
3.3.5 **Miscellaneous** Notification 26
3.4 Common Parameters 27
3.4.1 Event Type 27
3.4.2 Notification Object 28

3.4.3 Notifying Object . . . 28
3.4.4 Notification Class Identifier . . . 29
3.4.5 Notification Identifier . . . 29
3.4.6 Correlated Notifications . . . 30
3.4.7 Event Time . . . 30
3.4.8 Additional Text . . . 30
3.4.9 Additional Information . . . 30
3.5 Notification-Specific Parameters . . . 31
3.5.1 Alarm . . . 31
3.5.1.1 Probable Cause . . . 31
3.5.1.2 Specific Problems . . . 31
3.5.1.3 Perceived Severity . . . 31
3.5.1.4 Trend Indication . . . 32
3.5.1.5 Threshold Information . . . 32
3.5.1.6 Monitored Attributes . . . 32
3.5.1.7 Proposed Repair Actions . . . 32
3.5.2 State Change . . . 33
3.5.2.1 Source Indicator . . . 33
3.5.2.2 Changed State Attribute List . . . 33
3.5.2.2.1 Attribute Identifier . . . 33
3.5.2.2.2 Old Attribute Value . . . 33
3.5.2.2.3 New Attribute Value . . . 33
3.5.3 Object Creation/Deletion . . . 34
3.5.3.1 Source Indicator . . . 34
3.5.3.2 Attribute List . . . 34
3.5.3.2.1 Attribute Identifier . . . 34
3.5.3.2.2 Attribute Value . . . 34
3.5.4 Attribute Change . . . 34
3.5.4.1 Source Indicator . . . 35
3.5.4.2 Changed Attribute List . . . 35
3.5.4.2.1 Attribute Identifier . . . 35
3.5.4.2.2 Old Attribute Value . . . 35
3.5.4.2.3 New Attribute Value . . . 35
3.5.5 Security Alarm . . . 35
3.5.5.1 Security Alarm Cause . . . 35
3.5.5.2 Security Alarm Severity . . . 35
3.5.5.3 Security Alarm Detector . . . 36
3.5.5.4 Service User . . . 36
3.5.5.5 Service Provider . . . 36
3.5.6 Miscellaneous . . . 36
3.6 Notification Delivery Characteristics . . . 36
3.6.1 Discarded Notifications . . . 39
3.7 Filtering in the Subscriber API and Reader API . . . 40
3.8 Notification Suppression . . . 41
3.8.1 Static Notification Suppression . . . 43
3.9 Semantic Identification of Notification Elements . . . 44
3.10 Internationalization Issues . . . 45

3.11 API Design Goals . . . 46
3.12 Unavailability of the Notification Service API on a Non-Member Node . . . 46
3.12.1 A Member Node Leaves or Rejoins the Cluster Membership . . . 46
3.12.2 Guidelines for Notification Service Implementers . . . 47
3.13 Include File and Library Name . . . 48
3.14 Type Definitions . . . 48
3.14.1 Handles . . . 48
3.14.1.1 SaNtfHandleT . . . 48
3.14.1.2 SaNtfNotificationHandleT . . . 48
3.14.1.3 SaNtfNotificationFilterHandleT . . . 48
3.14.1.4 SaNtfReadHandleT . . . 48
3.14.2 SaNtfCallbacksT_3 . . . 49
3.14.3 SaNtfNotificationTypeT . . . 49
3.14.4 SaNtfEventTypeT . . . 49
3.14.5 SaNtfEventTypeBitmapT . . . 51
3.14.6 Notification Object . . . 51
3.14.7 Notifying Object . . . 52
3.14.8 SaNtfClassIdT . . . 52
3.14.9 SaServicesT . . . 52
3.14.10 SaNtfElementIdT . . . 53
3.14.11 SaNtfIdentifierT . . . 53
3.14.12 SaNtfCorrelationIdsT . . . 53
3.14.13 Event Time . . . 53
3.14.14 SaNtfValueTypeT . . . 54
3.14.15 SaNtfValueT . . . 55
3.14.16 Additional Text . . . 56
3.14.17 SaNtfAdditionalInfoT . . . 57
3.14.18 SaNtfProbableCauseT . . . 57
3.14.19 SaNtfSpecificProblemT . . . 59
3.14.20 SaNtfSeverityT . . . 59
3.14.21 SaNtfSeverityTrendT . . . 60
3.14.22 SaNtfThresholdInformationT . . . 60
3.14.23 SaNtfProposedRepairActionT . . . 61
3.14.24 SaNtfSourceIndicatorT . . . 61
3.14.25 SaNtfStateChangeT_3 . . . 61
3.14.26 SaNtfAttributeT . . . 62
3.14.27 SaNtfAttributeChangeT . . . 62
3.14.28 SaNtfServiceUserT . . . 62
3.14.29 SaNtfSecurityAlarmDetectorT . . . 63
3.14.30 SaNtfNotificationHeaderT . . . 64
3.14.31 SaNtfObjectCreateDeleteNotificationT . . . 65
3.14.32 SaNtfAttributeChangeNotificationT . . . 66
3.14.33 SaNtfStateChangeNotificationT_3 . . . 66
3.14.34 SaNtfAlarmNotificationT . . . 67
3.14.35 SaNtfSecurityAlarmNotificationT . . . 68
3.14.36 SaNtfMiscellaneousNotificationT . . . 68

3.14.37 Default Variable Notification Data Size 68
3.14.38 SaNtfSubscriptionIdT 69
3.14.39 SaNtfNotificationFilterHeaderT 69
3.14.40 SaNtfObjectCreateDeleteNotificationFilterT 70
3.14.41 SaNtfAttributeChangeNotificationFilterT 70
3.14.42 SaNtfStateChangeNotificationFilterT_2 71
3.14.43 SaNtfAlarmNotificationFilterT 72
3.14.44 SaNtfSecurityAlarmNotificationFilterT 73
3.14.45 SaNtfMiscellaneousNotificationFilterT 74
3.14.46 SaNtfSearchModeT 74
3.14.47 SaNtfSearchCriteriaT 74
3.14.48 SaNtfSearchDirectionT 75
3.14.49 SaNtfNotificationTypeFilterHandlesT_3 75
3.14.50 SaNtfNotificationsT_3 76
3.14.51 SaNtfStateT 76
3.14.52 SaNtfStaticFilterStateT 77
3.14.53 SaNtfSubscriberStateT 77
3.14.54 Limit Enumeration 77
3.14.55 SaNtfNotificationMinorIdT 77
3.15 Library Life Cycle 78
3.15.1 saNtfInitialize_3() 78
3.15.2 saNtfSelectionObjectGet() 80
3.15.3 saNtfDispatch() 82
3.15.4 saNtfFinalize() 83
3.16 Operations of the Producer API 85
3.16.1 saNtfObjectCreateDeleteNotificationAllocate() 86
3.16.2 saNtfAttributeChangeNotificationAllocate() 89
3.16.3 saNtfStateChangeNotificationAllocate_3() 91
3.16.4 saNtfAlarmNotificationAllocate() 94
3.16.5 saNtfSecurityAlarmNotificationAllocate() 96
3.16.6 saNtfMiscellaneousNotificationAllocate() 99
3.16.7 saNtfPtrValAllocate() 101
3.16.8 saNtfArrayValAllocate() 103
3.16.9 saNtfIdentifierAllocate() 105
3.16.10 saNtfNotificationSend() and saNtfNotificationSendWithId() 107
3.16.11 saNtfNotificationFree() 112
3.16.12 SaNtfStaticSuppressionFilterSetCallbackT_3 113
3.16.13 saNtfVariableDataSizeGet() 115
3.17 Consumer Operations 117
3.17.1 Common Consumer Operations 117
3.17.1.1 saNtfLocalizedMessageGet() 117
3.17.1.2 saNtfLocalizedMessageFree_2() 119
3.17.1.3 saNtfPtrValGet() 121
3.17.1.4 saNtfArrayValGet() 123
3.17.1.5 saNtfObjectCreateDeleteNotificationFilterAllocate() 125
3.17.1.6 saNtfAttributeChangeNotificationFilterAllocate() 127

3.17.2 saNtfStateChangeNotificationFilterAllocate_2() . . . 129
3.17.3 saNtfAlarmNotificationFilterAllocate() . . . 132
3.17.4 saNtfSecurityAlarmNotificationFilterAllocate() . . . 134
3.17.4.1 saNtfMiscellaneousNotificationFilterAllocate() . . . 137
3.17.4.2 saNtfNotificationFilterFree() . . . 139
3.17.5 Operations of the Subscriber API . . . 141
3.17.5.1 saNtfNotificationSubscribe_3() . . . 141
3.17.5.2 saNtfNotificationUnsubscribe_2() . . . 144
3.17.5.3 SaNtfNotificationCallbackT_3 . . . 146
3.17.5.4 SaNtfNotificationDiscardedCallbackT . . . 147
3.17.6 Operations of the Reader API . . . 149
3.17.6.1 saNtfNotificationReadInitialize_3() . . . 149
3.17.6.2 saNtfNotificationReadNext_3() . . . 153
3.17.6.3 saNtfNotificationReadFinalize() . . . 155

4 Notification Service UML Information Model . . . 157

4.1 DN Format for Notification Service UML Classes . . . 157
4.2 Notification Service UML Classes . . . 157

5 Notification Service Administration API . . . 161

5.1 Notification Service Administration API Model . . . 161
5.1.1 Notification Service Administration API Basics . . . 161
5.2 Include File and Library Name . . . 161
5.3 Type Definitions . . . 161
5.3.1 SaNtfAdminOperationIdT . . . 161
5.4 Notification Service Administration API . . . 162
5.4.1 SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER . . . 162
5.4.2 SA_NTF_ADMIN_DEACTIVATE_STATIC_SUPPRESSION_FILTER . . . 163

6 Alarms and Notifications . . . 165

6.1 Setting Common Attributes . . . 165
6.2 Notification Service Notifications . . . 166
6.2.1 Notification Service Alarms . . . 166
6.2.2 Notification Service State Change Notifications . . . 166
6.2.2.1 Static Suppression Filter Activated . . . 166
6.2.2.2 Static Suppression Filter Deactivated . . . 168
6.2.2.3 Subscriber Consuming Too Slowly . . . 169
6.2.2.4 Subscriber Consumes Fast Enough . . . 171

Appendix A Trap OID Mapping . . . 173

Appendix B Internationalization . . . 175

Appendix C API Usage Examples . . . 179

C.1 Producer Side (Example Function) – Object Create/Delete Notification 179
C.2 Producer Side (Example Function) – Attribute Change Notification 183
C.3 Producer Side (Example Function) – State Change Notification 186
C.4 Producer Side (Example Function) – Alarm Notification 189
C.5 Producer Side (Example Function) – Security Alarm Notification 194
C.6 Consumer Side (Example Function) – Subscribe for Notifications 197
C.7 Consumer Side (Example Function) – Read Logged Notifications 203

Index of Definitions 207

List of Figures

Figure 1: Entities Related to the Notification Service 22
Figure 2: UML Class Diagram 159

List of Tables

Table 1: Superseded Functions and Type Definitions in Version A.03.01 16
Table 2: Changes in Return Values of API and Administrative Functions 17
Table 3: Common Parameters of Alarms and Notification 27
Table 4: Specific Parameters of an Alarm 31
Table 5: Specific Parameters of a State Change Notification 33
Table 6: Specific Parameters of an Object Create Notification 34
Table 7: Specific Parameters of an Attribute Change Notification 34
Table 8: Specific Parameters of a Security Alarm 35
Table 9: Properties of Header Elements 108
Table 10: Properties of Elements in Object Create/Delete Notifications 108
Table 11: Properties of Elements in Attribute Value Change Notifications 109
Table 12: Properties of Elements in State Change Notifications 109
Table 13: Properties of Elements in Alarms 110
Table 14: Properties of Elements in Security Alarms 110
Table 15: DN Formats for Objects of Notification Service Classes 157
Table 16: Static Suppression Filter Activated Notification 167
Table 17: Static Suppression Filter Deactivated Notification 168
Table 18: Subscriber Consuming Too Slowly Notification 169
Table 19: Subscriber Consumes Fast Enough Notification 172
Table 20: Trap OID Mapping Table 173
Table 21: Internationalization Mapping Table 176
Table 22: Mapping Keywords to Notification Parameters 177

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 by application developers who use the Application Interface Specification to develop applications. The AIS 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).

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 ([1]).

1.3 History

Previous releases of the Notification Service specification:

(1) SAI-AIS-NTF-A.01.01

(2) SAI-AIS-NTF-A.02.01

This section presents the changes of the current release, SAI-AIS-NTF-A.03.01, with respect to the SAI-AIS-NTF-A.02.01 release. Editorial changes that do not change semantics or syntax of the described interfaces are not mentioned.

1.3.1 New Topics

⇒ According to the A.02.01 Notification Service specification, notification producers are alerted when an entire notification type is suppressed. The A.03.01 version of this specification has been changed to inform producers when all notifications of a particular event type are suppressed. This enhancement has led to the following changes to the specification:

- Replacement of the `SaNtfNotificationTypeBitsT` enumeration with the `SaNtfEventTypeBitmapT` enumeration (see Section 3.14.5 on page 51). This replacement has induced a few changes to Appendix C.

- Replacement of the `SaNtfStaticSuppressionFilterSetCallbackT` function with the `SaNtfStaticSuppressionFilterSetCallbackT_3` function (see Section 3.16.12 on page 113), as in the A.03.01 version the parameter `eventTypeBitmapT` is used instead of `notificationTypeBitmap`.
- As a consequence of the preceding change, the `SaNtfCallbacksT_3` type (see Section 3.14.2 on page 49) has replaced `SaNtfCallbacksT_2`. This change has also led to the replacement of the `saNtfInitialize_2()` function with `saNtfInitialize_3` (see Section 3.15.1 on page 78).
- The `saNtfFilterIsActive` attribute of the `SaNtfStaticFilter` (see Section 4.2 on page 157) class is now a persistent runtime attribute.

⇒ The miscellaneous notification type has been introduced to allow AIS Services and applications to generate notifications for events that do not directly map to the notification types provided in the A.02.01 version of the Notification Service. To support this new notification type, the following changes have been made to the specification:

- Definition of miscellaneous notification in the new Section 3.3.5 on page 26.
- Specification of possible values of the event type for the miscellaneous notification in Section 3.4.1 on page 27.
- Introduction of Section 3.5.6 on page 36 on specific parameters of the miscellaneous notification.
- Extension of Section 3.8 on page 41 and Section 3.8.1 on page 43 to state that notification suppression is also supported for a miscellaneous notification.
- Addition of a field to the `SaNtfNotificationTypeT` type definition (Section 3.14.3 on page 49), and addition of fields to the newly introduced `SaNtfEventTypeBitmapT` type (Section 3.14.5 on page 51).
- Extension of the `SaNtfEventTypeT` enumeration in Section 3.14.4 on page 49.
- Definition of the `SaNtfMiscellaneousNotificationT` type in Section 3.14.36 on page 68.
- Definition of the `SaNtfMiscellaneousNotificationFilterT` type in Section 3.14.45 on page 74.
- Replacement of the `SaNtfNotificationTypeFilterHandlesT` type with the `SaNtfNotificationTypeFilterHandlesT_3` type (see Section 3.14.49 on page 75) to include the miscellaneous filter handle.

- Replacement of the `SaNtfNotificationsT` type with the `SaNtfNotificationsT_3` type due to the introduction of a new field (see Section 3.14.50 on page 76). This modification has implied, in turn, the replacement of the `SaNtfNotificationCallbackT` function with the `SaNtfNotificationCallbackT_3` function (see Section 3.17.5.3 on page 146) and the replacement of the `saNtfNotificationReadNext()` function with the `saNtfNotificationReadNext_3()` function (see Section 3.17.6.2 on page 153.
- Definition of the `saNtfMiscellaneousNotificationAllocate()` function in Section 3.16.6 on page 99 to allocate memory for a miscellaneous notification.
- Definition of the `saNtfMiscellaneousNotificationFilterAllocate()` function in Section 3.17.4.1 on page 137 to allocate memory for a miscellaneous notification filter.
- Replacement of the `saNtfNotificationSubscribe()` function with the `saNtfNotificationSubscribe_3` function (see Section 3.17.5.1 on page 141) due to the previously mentioned replacement of the `SaNtfNotificationTypeFilterHandlesT` type with the `SaNtfNotificationTypeFilterHandlesT_3` type.
- Replacement of the `saNtfNotificationReadInitialize_2()` function with the `saNtfNotificationReadInitialize_3` function (see Section 3.17.6.1 on page 149) due to the previously mentioned replacement of the `SaNtfNotificationTypeFilterHandlesT` type with the `SaNtfNotificationTypeFilterHandlesT_3` type.
- Definition of the `SaNtfMiscellaneousFilterElementSet` object class, shown in FIGURE 2 in Section 4.2.

⇒ The handling of correlation identifiers has been extended. This extension has led to the following changes to the specification:

- Note on the notification identifier in Section 3.4.5 on page 29.
- Introduction of root and parent events in Section 3.4.6 on page 30.
- New Section 3.14.12 on page 53 on the `SaNtfCorrelationIdsT` type.
- Extension of Section 3.16 on page 85 on the operations of the producer API.
- Definition of the `saNtfIdentifierAllocate()` function in Section 3.16.9 on page 105 to allocate a notification identifier.

- Definition of the `saNtfNotificationSendWithId()` function in Section 3.16.10 on page 107 to send a notification with the notification identifier obtained by a previous invocation of the `saNtfIdentifierAllocate()` function. Several places of the specification containing references to the `saNtfNotificationSend()`function now refer additionally to the `saNtfNotificationSendWithId()` function.

⇒ The `SaNtfNotificationMinorIdT` type (see Section 3.14.55) has been introduced to specify the `minorId` field of notifications produced by the Notification Service. The descriptions of the notifications in Section 6.2 refer to this type.

⇒ The types of the `oldState` and `newState` fields of the `SaNtfStateChangeT` type have changed to `SaUint64`. As a consequence, the `SaNtfStateChangeT` type has been replaced with the `SaNtfStateChangeT_3` type (see Section 3.14.25 on page 61). This modification implied, in turn, the replacement of the `SaNtfStateChangeNotificationT` type with the `SaNtfStateChangeNotificationT_3` type (see Section 3.14.33 on page 66) and the replacement of the `saNtfStateChangeNotificationAllocate()` function with the `saNtfStateChangeNotificationAllocate_3()` function (see Section 3.16.3 on page 91).

1.3.2 Clarifications

None

1.3.3 Deleted Topics

- Section 2.2 and Chapter 7 of the Notification Service specification A.02.01, which described the SNMP MIBs interface, have been removed, as no MIBs are provided in the Notification Service specification A.03.01.
- Section 3.9 in the A.02.01 Notification Service specification on the integration of HPI events (see [9]) has been removed, as it is now part of the PLM Service specification (see [8]).

1.3.4 Other Changes

- The default value (AMF) for the source indicator element in the state change notification (see Section 3.5.2 on page 33) has been removed.
- The type for the `saNtf<type>NotificationClassIds` attributes has been changed from `SaNtfClassIdT` to `SaStringT` in the UML classes `SaNtfObjectCreateDeleteFilterElementSet`, `SaNtfAttributeChangeFilterElementSet`, and `SaNtfStateChangeFilterElementSet`, shown in the UML diagram in Section 4.2. Section 4.2 also provides a rule to convert the values of a `SaNtfClassIdT` structure to a `SaStringT` type.
  Note that the `SaStringT` type is also used for the `saNtfMiscNotificationClassIds` attribute in the newly introduced `SaNtfMiscellaneousFilterElementSet`.
- In Section 5.3.1 on page 161, a typo in the name of the `saNtfAdminOperationIdT` type definition has been corrected, as it should be `SaNtfAdminOperationIdT`.

1.3.5 Superseded and Superseding Functions

The Notification Service defines for the version A.03.01 new functions and new type definitions to replace functions and type definitions of the version A.02.01. The list of replaced functions and type definitions in alphabetic order is presented in Table 1.

The superseded functions and type definitions are no longer supported in version A.03.01, and no description is provided for them in this document. The names of the superseding functions and type definitions are obtained by adding "_3" to the respective names of the A.02.01 version or by replacing "_2" by "_3" if the superseded functions or type definitions had already "_2" at the end of their names. Exceptions to this rule are indicated by table footnotes. Regarding the support of backward compatibility in SA Forum AIS, refer to [2].

Table 1 Superseded Functions and Type Definitions in Version A.03.01

Functions and Type Definitions of Version A.02.01 no Longer Supported in A.03.01
`SaNtfCallbacksT_2`
`saNtfInitialize_2()`
`SaNtfNotificationCallbackT`
`saNtfNotificationReadInitialize_2()`
`saNtfNotificationReadNext()`
`SaNtfNotificationsT`
`saNtfNotificationSubscribe()`
`SaNtfNotificationTypeBitsT`[1]
`SaNtfNotificationTypeFilterHandlesT`
`saNtfStateChangeNotificationAllocate()`
`SaNtfStateChangeNotificationT`
`SaNtfStateChangeT`
`SaNtfStaticSuppressionFilterSetCallbackT`

1. Replaced with `SaNtfEventTypeBitmapT`

1.3.6 Changes in Return Values of API and Administrative Functions

The following table applies only to functions that have not been superseded.

Table 2 Changes in Return Values of API and Administrative Functions

Function	Return Value	Change Type
All administrative operations described in Chapter 5	`SA_AIS_ERR_TIMEOUT` `SA_AIS_ERR_NO_MEMORY`	new
All API functions except `saNtfFinalize()` and `SaNtfNotificationDiscardedCallbackT`	`SA_AIS_ERR_UNAVAILABLE`	extended
`saNtfAttributeChangeNotificationFilterAllocate()`	`SA_AIS_ERR_BAD_HANDLE`	documentation fix [1]

1. SAI-AIS-NTF-A.02.01 referred to `notificationHandle`. It has been corrected to refer to `ntfHandle`.

1.4 References

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

[1] Service Availability™ Forum, Service Availability Interface, Overview, SAI-Overview-B.05.01

[2] Service Availability™ Forum, Service Availability Interface, C Programming Model, SAI-AIS-CPROG-B.05.01

[3] Service Availability™ Forum, Application Interface Specification, Information Model Management Service, SAI-AIS-IMM-A.03.01

[4] Service Availability™ Forum, Application Interface Specification, Cluster Membership Service, SAI-AIS-CLM-B.04.01

[5] Service Availability™ Forum, SA Forum Information Model in XML Metadata Interchange (XMI) v2.1 format, SAI-IM-XMI-A.04.01.xml.zip

[6] Service Availability™ Forum, Application Interface Specification, Availability Management Framework, SAI-AIS-AMF-B.04.01

[7] Service Availability™ Forum, Application Interface Specification, Event Service, SAI-AIS-EVT-B.03.01

[8] Service Availability™ Forum, Application Interface Specification, Platform Management Service, SAI-AIS-PLM-A.01.01

[9] Service Availability™ Forum, Hardware Platform Interface Specification, SAI-HPI-B.03.01

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

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

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

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

[14] SNMP, SNMPv2, SNMPv3, and RMON 1 and 2
William Stallings. 1999. Addison Wesley

[15] 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 Web site (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 SA Forum Web site (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 SA Forum Web site (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 SA Forum Web site (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. Normative references to ITU-T-defined agents and managers are used in the definition of the current notification standard.

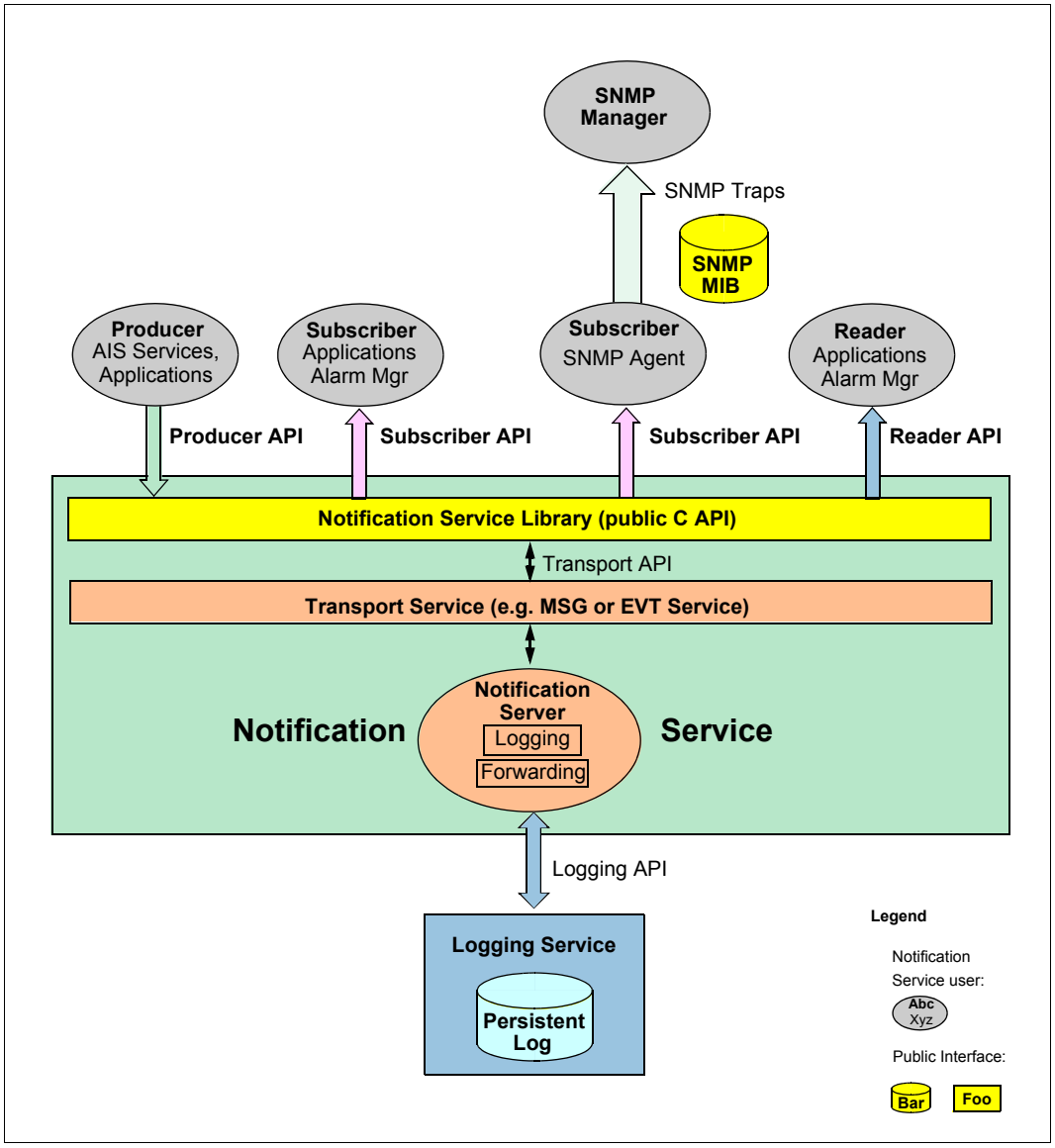
Adapting the definition from ITU-T X.710 to the present SA Forum context led to the following definition:

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.

Note: In the preceding sentence and in the remainder of this document, the term “event” has the same meaning as in commonly understood English—an incident, or simply, a change of status. This avoids confusion with the Event Service (which is specified in [7]), this document defines the term `notification`.

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

FIGURE 1 Entities Related to the Notification Service



2.1 Users of the Notification Service Library

Users of the Notification Service library run on cluster nodes as defined by the Cluster Membership Service (see [4]).

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 a consumer is not interested in all notifications, it can specify filter criteria. A consumer can also be a producer. A consumer can be one of the types described in the following subsections or both:

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 Notification Service

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

2.2.1 Notification Service Library

The Notification Service library provides the following public C APIs:

- Producer API
- Subscriber API
- Reader API

2.2.2 Notification Server

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

2.2.3 Transport Service

The transport service links the Notification Service library and the notification server. It is currently not specified.

2.3 Log 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 Log Service to write notifications into persistent storage. Likewise, the Notification Reader API may also use the log files of the Log Service.

3 Notification Service API

3.1 Notifications

Notifications are 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 API**. 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, a notification (or an 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 an SA Forum cluster:

- Alarm
- State change
- Object create/delete
- Attribute change
- Security alarm
- Miscellaneous

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 ([12]).

In the context of the SA Forum, Application Interface Specification (AIS) Services, Frameworks, applications, Hardware Platform Interface (HPI) listener, and proxies for non-SA applications can send alarm notifications. An application detecting a communication failure, an operating system reaching some threshold for the maximum number of files opened, and AIS Services or the Availability Management Framework (AMF) running into internal errors are 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 from either the internal operation of the managed object or a management operation. This is defined in X.731 ([11]).

Example: The changes of presence state, readiness state, and HA state of service units and components performed by the Availability Management Framework can be reported to management applications by using these notifications.

3.3.3 Object Create/Delete and Attribute Change Notifications

The **object create/delete** and **attribute change** notifications report creation and deletion of managed objects and attribute changes of configuration data on a managed object. This is defined in X.730 ([10]).

Example: Application-specific information like threads created, users added or deleted, modifications to default configuration data, and so on, can be reported to management applications by using these notifications.

3.3.4 Security Alarm Notification

The **security alarm** notification is used to report an event indicating that an attack or potential attack on system security has been detected. This is defined in X.736 ([13]).

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

3.3.5 Miscellaneous Notification

The **miscellaneous** notification type is introduced to allow AIS Services and applications to generate notifications for events that do not directly map to the previous notification types. Example of events that may be reported using the miscellaneous notification type include HPI events, administrative operations performed on IMM objects, and application specific events.

3.4 Common Parameters

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

Table 3 Common Parameters of Alarms and 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	–
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

The **event type** depends on the type of notifications.

For alarm notifications, the event type 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 create/delete and attribute change notifications 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 the managed hardware;
- security service violation: security service has detected a security attack;
- time domain violation: an event has occurred at an unexpected or prohibited time.

Possible values of the event type for a miscellaneous notification are:

- administrative operation start/end events;
- error report/clear events;
- HPI events related to resources, sensors, diagnostic, firmware upgrade, …;
- application specific events.

It is possible to derive the notification type from the event type parameter.

3.4.2 Notification Object

The **notification object** is a logical entity that is identified by its LDAP DN and about which the notification is generated.

3.4.3 Notifying Object

The **notifying object** is a logical entity that is identified by its LDAP DN and that is sending the notification.

3.4.4 Notification Class Identifier

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

Notifications 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 notification classes. (Note that the term “class” is used here 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.)

Examples for notification classes, taken from the AIS:

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

In the first example, the aforementioned varying parameter values could be the notification object (the message queue name); in the second example, they could be the notification object (the service instance name) and the attribute list (the 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 an SI name, and the second parameter will always be an SU name.

For identification purposes, each notification class is assigned a unique numeric **notification class identifier** (**NCI**). To avoid conflicts among identifier values from different vendors and applications, the notification class identifier is explicitly divided into a vendor identifier and a vendor-specific part.

3.4.5 Notification Identifier

The **notification identifier** parameter, if present, provides an identifier for a specific notification instance. The notification identifier is generated by the Notification Service; its value is unique within the cluster, but not necessarily cluster-wide monotonically increasing. The notification identifier may be carried in the correlated notifications parameter (see Section 3.4.6) of future notifications.

3.4.6 Correlated Notifications

Correlated notifications are a set of notifications generated earlier and that are 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.

Some events occurring in an SA Forum cluster such as error reports or administrative operations may trigger extensive changes in the cluster and hence generate a large number of related notifications. In such situations, it would be too expensive to mandate that each notification must hold the list of all correlated notifications generated earlier. SA Forum defines a simpler scheme: typically, a root event triggers a set of other events, each of which triggers itself a set of other events, and so on. Thus, notifications related to these events can be organized in a tree structure. To help to reconstruct afterwards the tree of correlated notifications, all AIS Services should include, whenever possible, the root and parent notification in the notification tree structure as correlated notifications for each notification they generate.

3.4.7 Event Time

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

3.4.8 Additional Text

The **additional text** field allows a free form text description to be reported.

3.4.9 Additional Information

The **additional information** is a data structure that may carry more data not covered by the standard fields in the report. This parameter is opaque, and it 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:

Table 4 Specific Parameters of an Alarm

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

The **probable cause** 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

Specific problems is a further refinement to the probable cause field.

3.5.1.3 Perceived Severity

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

- **Cleared**: This means that a previously reported alarm is cleared. Clearing of alarms can be done based on matching event types, probable cause, and specific problem. It may also 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; however, 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

The **trend indication** is important when a logical entity has already 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, the **threshold information** may be used to capture that information. Threshold information encapsulates the threshold identifier, the actual threshold value, the threshold hysteresis (important to avoid repeated alarms), the observed value of the parameter, and the time of last threshold crossing.

3.5.1.6 Monitored Attributes

The **monitored attributes** field is useful in reporting any changing attributes of the logical entity that may be of interest in relation to this alarm. This field uses the same syntax as the attribute list in object creation/deletion notifications (see Section 3.5.3.2).

3.5.1.7 Proposed Repair Actions

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

3.5.2 State Change

Specific parameters for this report are:

Table 5 Specific Parameters of a State Change Notification

Name	X.73x-Recommendation	Default Value
Source Indicator	Optional Parameter	–
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

The **source indicator** 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

The **changed state attribute list** is a list of attribute identifiers. Multiple types of state changes (for instance, life cycle, readiness, and HA state) 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

The **attribute identifier** is an identifier for the state attribute that is being modified, for instance, life cycle, readiness, and HA state.

3.5.2.2.2 Old Attribute Value

The **attribute value** is the value of the state attribute before the change.

3.5.2.2.3 New Attribute Value

The **new attribute value** is the value of the state attribute after the change.

3.5.3 Object Creation/Deletion

Specific parameters for this report are:

Table 6 Specific Parameters of an Object Create Notification

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 the source indicator defined in Section 3.5.2.1.

3.5.3.2 Attribute List

The **attribute list** is a list of attributes along with and their current values at the time the logical entity was created or deleted.

3.5.3.2.1 Attribute Identifier

The **attribute identifier** is an identifier for an attribute.

3.5.3.2.2 Attribute Value

The **attribute value** is the value of the attribute at the time of creation/ deletion.

3.5.4 Attribute Change

Specific parameters for this report are:

Table 7 Specific Parameters of an Attribute Change Notification

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 the source indicator defined in Section 3.5.2.1.

3.5.4.2 Changed Attribute List

The **changed attribute list** 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

The **attribute identifier** is an identifier for the attribute that is being modified.

3.5.4.2.2 Old Attribute Value

The **old attribute value** is the value of the attribute before the change.

3.5.4.2.3 New Attribute Value

The **new attribute value** is the value of the attribute after the change.

3.5.5 Security Alarm

Specific parameters for this report are:

Table 8 Specific Parameters of a Security Alarm

Name	X.73x-Recommendation	Default Value
Cause	Mandatory Parameter	–
Severity	Mandatory Parameter	–
Detector	Mandatory Parameter	–
Service User	Mandatory Parameter	–
Service Provider	Mandatory Parameter	–

3.5.5.1 Security Alarm Cause

The **security alarm cause** field is similar to the probable cause field in alarm notifications (see Section 3.5.1.1). A list of generic severity alarm causes is given in the X.736 standard.

3.5.5.2 Security Alarm Severity

The **security alarm severity** field is the same as the perceived severity field in alarm notifications (See Section 3.5.1.3).

3.5.5.3 Security Alarm Detector

The **security alarm detector** 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 **service provider** field indicates the intended service provider of the service that led to this security alarm.

3.5.6 Miscellaneous

A miscellaneous notification contains only the common parameters, and no specific parameters are defined for it.

3.6 Notification Delivery Characteristics

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

- **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 specify the guaranteed delivery in more detail.
  - If the producer fails while it (or one thread of it) is calling `saNtfNotificationSend()` or `saNtfNotificationSendWithId()`, the notification is forwarded either to all subscribers or to no subscriber.
    Note that it is not intended to block the call to `saNtfNotificationSend()` or `saNtfNotificationSendWithId()` until the notification is forwarded to all subscribers; instead, 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 Service library fails to forward a produced notification, the Notification Service 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, the process of forwarding is completed when this server process has been either 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 while it is forwarding the notification.
- If an implementation of the Notification Service has one or more instances of separate server processes, and a notification is 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 Log Service, the instance that does the forwarding to the Log Service (depending on the implementation, this instance could be either the Notification Service 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 Log 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, it need not call `saNtfDispatch()`. Under these conditions, retry attempts might occur only when the application calls `saNtfNotificationSend()` or `saNtfNotificationSendWithId()` 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 Notification Service invokes the subscriber's `SaNtfNotificationDiscardedCallbackT` callback function to inform the subscriber about the notifications that could not be delivered to it. 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 using 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 (that is, `SaNtfNotificationCallbackT_3`). This enables the subscriber to get all of these notifications (that is, 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 retransmission 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 (that is, `SaNtfNotificationCallbackT_3`), these remaining notifications are implicitly discarded by the Notification Service. It is the responsibility of the subscriber to checkpoint on the delivered notifications. When the subscriber is restarted after failure, or if it 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 cancelation also implies that no new notifications can be forwarded to the failed subscriber before it restarts and subscribes again.
- If a cluster node on which a notification is being forwarded leaves the cluster membership (see [4]), the delivery of the notification to subscribers and to the Log Service is not guaranteed.

- **At most once delivery**
  The Notification Service must not deliver a notification to the same subscription of a process (previously installed in an invocation of the `saNtfNotificationSubscribe_3()` function) multiple times.
- **Ordering**
  For a given notification type, the notifications are received by subscribers in the same order as they were generated by the producer. Likewise, when a user of the Reader API reads logged notifications in chronological order, the user 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 multiple producers generate notifications at the same time, it is not predictable in which order they will arrive at the subscriber. Under certain conditions, for instance, 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 also true for discarded notifications, that is, the invocation of the 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. For example, if the producer crashes while it (or one thread of it) is calling saNtfNotificationSend() or saNtfNotificationSendWithId(), either the complete notification or no notification is forwarded to the subscribers. Note that due to the limited record size of the Log Service, data truncation may occur when notifications are logged. This may happen when an implementation of the Notification Service allows for notifications with very large data while the record size of the Log Service in use has set the record size for notifications to a small value. In this case, it is implementation-specific which parts of a notification are omitted when the notification is read with the Reader API.
- **Persistence**
  Alarm notifications and security alarm notifications must be stored persistently (whereas object creation/deletion, attribute value change and state change notifications need not be stored persistently). The term “persistently” means here that the stored notifications will still be available after a cluster reboot. The Reader API allows to retrieve the logged notifications. Retrieving these logged notifications is particularly important for some of the aforementioned 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 passed to it by invoking SaNtfNotificationCallbackT_3.

(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 notify the subscriber about discarded notifications. The Notification Service invokes the subscriber’s SaNtfNotificationDiscardedCallbackT callback function for this purpose. When this callback is invoked, the subscriber may recover the notifications

either 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 no longer exists. After restart or fail-over to another process, the new subscriber process can synchronize with the list of notifications by 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 when it unsubscribes will do no harm. Otherwise, if the intention is actually to change filter criteria of a subscription, the subscriber should first subscribe with the new filter criteria and then unsubscribe from the previous subscription (with the old filter criteria).

3.7 Filtering in the Subscriber API and Reader API

The Notification Service defines notification-type-specific functions used to allocate filters. Functions of the Subscriber API and Reader API take an `SaNtfNotificationTypeFilterHandlesT_3` parameter (see Section 3.14.49 on page 75), which contains handles for all notification-type-specific filters. (Filter handles are returned by the filter allocation functions.) For each notification type in which the caller is interested, filter criteria have to be specified (that is, a filter handle has to be set in the `SaNtfNotificationTypeFilterHandlesT_3` 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, and so on. Each such filter element can be a set of explicit values or an empty set. If not empty, 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, 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.

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 kind of filtering is termed **low level filtering**, since the meaning of a particular value of a filter element is not interpreted. However, for notification object and notifying object filter elements, **high level filtering** is also provided in these two specific cases:

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

Note that the filtering for notification object and notifying object described above does not create a dependency on the Availability Management Framework. The Availability Management Framework 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.

3.8 Notification Suppression

This section discusses the suppression of notifications.

The mechanism of **notification suppression** is essential to avoid situations where floods of unimportant/dispensable notifications are generated. Notification suppression is supported for all types of notifications except for alarms and security alarms.

Typical reasons why it makes sense to have a suppression mechanism for notifications are:

- Notification floods may contain many notifications reflecting only minor changes in the system.
- 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. Under such circumstances, 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 production system.

The above described situations for notification floods need different handling; in the first case, for example, suppression of all notifications about those minor changes may be needed; in the third 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 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.

Note that this version of the Notification Service does not specify the dynamic suppression of notifications. However, such a mechanism may be part of a future version of this specification.

The Notification Service provides **close-to-source** suppression, which 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 Service library suppresses notifications matching the current suppression settings.

3.8.1 Static Notification Suppression

For static suppression of notifications, filters can be configured. The filters are defined as UML classes (for details on these classes, refer to Chapter 4).

A filter contains a list of one or more filter element sets. Specific filter element sets are defined for all notification types that can be suppressed:

- object create/delete notification type
- attribute value change notification type
- state change notification type
- miscellaneous notification type

A filter element set contains filter elements. A filter element is related to an attribute of a notification; for example, the notification attribute `notificationClassId` can also be used as a filter element.

Each filter element consists of a list of explicit values; the value list can be empty.

If not empty, the filter element matches for a particular notification if one of the specified values match. This means that for one filter element all values in the list are logically ORed. If a filter element has an empty value list, then all values of a notification for that particular element match (pass through).

A notification matches the filter element set if it matches all filter elements in the filter element set. This means that all filter elements of a filter element set are logically ANDed.

A notification matches a filter if it matches at least one filter element set. This means that within a filter all filter element sets are logically ORed.

Administrative operations are defined to activate and to deactivate a filter for static suppression (for information on administrative operations, refer to Chapter 5). When no filter is active, none of the notifications are suppressed. When one or more filters are active, only those notifications that match any of the active filters are suppressed. This means that filters are logically ORed.

The following two special cases are of important practical relevance.

- When one or more filters are active, but none of them contains a filter element set for a particular notification type, notifications of this notification type are not suppressed. For example, if there is currently one active filter and this filter does

not contain any filter element set for object create/delete notifications, such notifications are not suppressed.

- When a filter is active, and it contains at least one filter element set for a particular notification type, and all filter elements in the filter element set have empty value lists, all notifications of this notification type are suppressed. For example, if a currently active filter has one filter element set for attribute value change notification, and within this filter element set all value lists are empty, all attribute value change notifications match the filter and are, therefore, suppressed.

3.9 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, and 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, so that subscribers can 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 Section 3.14.19) need a special handling concerning the notification element identifier.

3.10 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. Human readers prefer a concise textual description of the situation in the human language of their choice. 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 achieved here by using the notification class identifier as a starting point, and by defining a specific text for each notification class identifier. To provide a concise textual description of the situation, each specific text may then refer to those notification parameters that are most important for describing that situation. The detailed syntax can be found in Appendix B on page 175.

Note that the internationalization mechanism provided by the Notification Service does not translate any of the notification parameter values (for instance, the additional text parameter or other character string parameters); instead, it provides a link between a notification class identifier and localized text related to that notification class identifier. However, the localized text can contain variable parts that are references to notification parameter values.

3.11 API Design Goals

The API of the Notification Service was designed to meet the following goals:

- 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 and 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, several attributes are in fact arrays of variable length; for some of them, each array element is even 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 are defined for notifications and notification filters, which makes 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), that is, 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.

3.12 Unavailability of the Notification Service API on a Non-Member Node

The Notification Service does not provide service to processes on cluster nodes that are not in the cluster membership (see [4]).

The following subsection describes the behavior of the Notification Service under various conditions that cause the Notification Service to be unavailable on a node. Section 3.12.2 contains guidelines to Notification Service implementers for dealing with a temporary unavailability of the service.

3.12.1 A Member Node Leaves or Rejoins the Cluster Membership

If the cluster node has left the cluster membership (see [4]) or is being administratively evicted from the cluster membership, the Notification Service behaves as follows towards processes residing on that node and using or attempting to use the service:

- Calls to `saNtfInitialize_3()` will fail with `SA_AIS_ERR_UNAVAILABLE`.
- All Notification Service APIs that are invoked by the process and that operate on handles already acquired by the process will fail with `SA_AIS_ERR_UNAVAILABLE` with the exception of `saNtfFinalize()`, which is used to free the library handles and all resources associated with these handles.
- All callbacks (`SaNtfNotificationCallbackT_3`, `SaNtfNotificationDiscardedCallbackT`, and `SaNtfStaticSuppressionFilterSetCallbackT_3`) will not be called.

If the cluster node rejoins the cluster membership, processes executing on the cluster node will be able to reinitialize new library handles and use the entire set of Notification Service APIs that operate on these new handles; however, invocation of APIs that operate on handles acquired by any process before the cluster node left the membership will continue to fail with `SA_AIS_ERR_UNAVAILABLE` with the exception of `saNtfFinalize()`, which is used to free the library handles and all resources associated with these handles. Hence, it is recommended for the processes to finalize the library handles as soon as the processes detect that the cluster node left the membership.

When the cluster node leaves the membership, the Notification Service executing on the remaining nodes of the cluster behaves as if all processes that were using the Notification Service on the leaving node had been terminated.

3.12.2 Guidelines for Notification Service Implementers

The implementation of the Notification Service must leverage the SA Forum Cluster Membership Service (see [4]) to determine the membership status of a cluster node for the case explained in Section 3.12.1 before returning `SA_AIS_ERR_UNAVAILABLE`. If the Cluster Membership Service considers a cluster node as a member of the cluster but the Notification Service experiences difficulty in providing service to its clients because of transport, communication, or other issues, it must respond with `SA_AIS_ERR_TRY_AGAIN`.

3.13 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
```

3.14 Type Definitions

3.14.1 Handles

3.14.1.1 SaNtfHandleT

```
typedef SaUint64T SaNtfHandleT;
```

This type is used for the handle that the Notification Service provides to a process during initialization of the Notification Service library and that is used by the process when it invokes functions of the Notification Service API.

3.14.1.2 SaNtfNotificationHandleT

```
typedef SaUint64T SaNtfNotificationHandleT;
```

This type is used for a handle to the internal notification structure that is used in API calls.

3.14.1.3 SaNtfNotificationFilterHandleT

```
typedef SaUint64T SaNtfNotificationFilterHandleT;
```

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

3.14.1.4 SaNtfReadHandleT

```
typedef SaUint64T SaNtfReadHandleT;
```

This type is used for a handle that is used in the Reader API.

3.14.2 SaNtfCallbacksT_3

A structure of the SaNtfCallbacksT_3 type (called a callbacks structure) is used to specify the callback functions that the Notification Service can invoke.

```
typedef struct {
SaNtfNotificationCallbackT_3
      saNtfNotificationCallback;
SaNtfNotificationDiscardedCallbackT
      saNtfNotificationDiscardedCallback;
SaNtfStaticSuppressionFilterSetCallbackT_3
      saNtfStaticSuppressionFilterSetCallback;
} SaNtfCallbacksT_3;
```

3.14.3 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,
      SA_NTF_TYPE_MISCELLANEOUS             = 0x6000
} SaNtfNotificationTypeT;
```

This is the enumeration of all notification types.

3.14.4 SaNtfEventTypeT

```
#define SA_NTF_NOTIFICATIONS_TYPE_MASK          0xF000
```

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

```
/* Generic event 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,
    /* other event types supported */
        SA_NTF_MISCELLANEOUS_NOTIFICATIONS_START =
             SA_NTF_TYPE_MISCELLANEOUS,
        SA_NTF_APPLICATION_EVENT,
        SA_NTF_ADMIN_OPERATION_START,
        SA_NTF_ADMIN_OPERATION_END,
        SA_NTF_CONFIG_UPDATE_START,
        SA_NTF_CONFIG_UPDATE_END,
        SA_NTF_ERROR_REPORT,
        SA_NTF_ERROR_CLEAR,
        SA_NTF_HPI_EVENT_RESOURCE,
        SA_NTF_HPI_EVENT_SENSOR,
        SA_NTF_HPI_EVENT_WATCHDOG,
        SA_NTF_HPI_EVENT_DIMI,
        SA_NTF_HPI_EVENT_FUMI,
        SA_NTF_HPI_EVENT_OTHER
    } SaNtfEventTypeT;
```

`SaNtfEventTypeT` defines all event types that are allowed in notifications.

3.14.5 SaNtfEventTypeBitmapT

```
#define SA_NTF_OBJECT_CREATION_BIT                0x01
#define SA_NTF_OBJECT_DELETION_BIT                0x02
#define SA_NTF_ATTRIBUTE_ADDED_BIT                0x04
#define SA_NTF_ATTRIBUTE_REMOVED_BIT              0x08
#define SA_NTF_ATTRIBUTE_CHANGED_BIT              0x10
#define SA_NTF_ATTRIBUTE_RESET_BIT                0x20
#define SA_NTF_OBJECT_STATE_CHANGE_BIT            0x40
#define SA_NTF_ALARM_COMMUNICATION_BIT            0x80
#define SA_NTF_ALARM_QOS_BIT                      0x100
#define SA_NTF_ALARM_PROCESSING_BIT               0x200
#define SA_NTF_ALARM_EQUIPMENT_BIT                0x400
#define SA_NTF_ALARM_ENVIRONMENT_BIT              0x800
#define SA_NTF_INTEGRITY_VIOLATION_BIT            0x1000
#define SA_NTF_OPERATION_VIOLATION_BIT            0x2000
#define SA_NTF_PHYSICAL_VIOLATION_BIT             0x4000
#define SA_NTF_SECURITY_SERVICE_VIOLATION_BIT     0x8000
#define SA_NTF_TIME_VIOLATION_BIT                 0x10000
#define SA_NTF_ADMIN_OPERATION_START_BIT          0x20000
#define SA_NTF_ADMIN_OPERATION_END_BIT            0x40000
#define SA_NTF_CONFIG_UPDATE_START_BIT            0x80000
#define SA_NTF_CONFIG_UPDATE_END_BIT              0x100000
#define SA_NTF_ERROR_REPORT_BIT                   0x200000
#define SA_NTF_ERROR_CLEAR_BIT                    0x400000
#define SA_NTF_HPI_EVENT_RESOURCE_BIT             0x800000
#define SA_NTF_HPI_EVENT_SENSOR_BIT               0x1000000
#define SA_NTF_HPI_EVENT_WATCHDOG_BIT             0x2000000
#define SA_NTF_HPI_EVENT_DIMI_BIT                 0x4000000
#define SA_NTF_HPI_EVENT_FUMI_BIT                 0x8000000
#define SA_NTF_HPI_EVENT_OTHER_BIT                0x10000000
#define SA_NTF_APPLICATION_EVENT_BIT              0x100000000000

typedef SaUint64T SaNtfEventTypeBitmapT;
```

The `SaNtfEventTypeBitmapT` type is a bitmap of event types. The bit representing a particular event type in the bitmap is specified by the listed values.

3.14.6 Notification Object

The type `SaNameT` is used for the notification object. The notification object will typically be LDAP DNs as defined by AIS, for instance, for a component of the Availability Management Framework, for a message queue of the Message Service, or for 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 applica-

tion-specific objects. The value of the notification object is interpreted by the Notification Service only for those cases defined in Section 3.7 on page 40.

3.14.7 Notifying Object

The type `SaNameT` is used for the notifying object. The notifying object will typically be the LDAP DN of an Availability Management Framework logical entity producing the notification. If the notifying object is not an Availability Management Framework logical entity, an application-specific notation may be used instead. Currently, the Notification Service does not define a naming scheme for notifying objects that are no Availability Management Framework components. The value of the notification object is interpreted by the Notification Service only for those cases defined in Section 3.7 on page 40.

3.14.8 SaNtfClassIdT

```
typedef struct {
     SaUint32T vendorId;
     SaUint16T majorId;
     SaUint16T minorId;
} SaNtfClassIdT;
```

This type 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 [15]. The `majorId` and `minorId` values can be arbitrarily assigned to a notification class identifier by a vendor.

```
#define SA_NTF_VENDOR_ID_SAF 18568
```

This is a predefined `vendorId` for those notification class identifiers specified by SA Forum. The SNMP enterprise number of SA Forum is used here. For the predefined values of `majorId` of the SA Forum Services, see `SaServicesT` in [2].

3.14.9 SaServicesT

This enumeration type is defined in [2]. It specifies the values for the SA Forum Services as used for `majorId` in `SaNtfClassIdT`, that is, the values used for `majorId` when `vendorId` is `SA_NTF_VENDOR_ID_SAF`.

3.14.10 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).

3.14.11 SaNtfIdentifierT

```
typedef SaUint64T SaNtfIdentifierT;
```

This type is used for notification identifiers.

```
#define SA_NTF_IDENTIFIER_UNUSED((SaNtfIdentifierT) 0LL)
```

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.

3.14.12 SaNtfCorrelationIdsT

As described in Section 3.4.6, AIS Services should provide the notification identifiers of the root and parent notifications for each notification they generate. The SaNtfCorrelationIdsT type is used to pass the required notification identifiers to be used as correlation identifiers in related notifications.

```
typedef struct {
     SaNtfIdentifierT rootCorrelationId;
     SaNtfIdentifierT parentCorrelationId;
     SaNtfIdentifierT notificationId;
} SaNtfCorrelationIdsT;
```

If the invoker of the API has only a rootCorrelationId and no parentCorrelationId, it must set the parentCorrelationId to the rootCorrelationId value. If the invoker of the APIs does not have any correlation identifier to provide, it must set rootCorrelationId and parentCorrelationId to SA_NTF_IDENTIFIER_UNUSED.

Some APIs may use the notificationId field to return the notification identifier of the first notification that has been sent by the invoked service as a consequence of the API invocation.

3.14.13 Event Time

The type SaTimeT is used for the event time.

3.14.14 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;
```

The `SaNtfValueTypeT` type defines the possible types of the values within a structure of type `SaNtfValueT`.

3.14.15 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 be used only 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 having identical type *
      * like notification identifiers, attributes, and so on.*
      * It may only be used in conjunction with the data type value *
      * SA_NTF_VALUE_ARRAY. The 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;
```

The SaNtfValueT type defines a structure that is used in notifications for parameters or parameter elements that may be of varying data type. A value can 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, such as SA_NTF_VALUE_STRING or SA_NTF_VALUE_ARRAY, SaNtfValueT cannot hold the memory needed to store the actual data; for these data types, additional memory has to be reserved outside SaNtfValueT. This memory is allocated by invoking either saNtfPtrValAllocate() or saNtfArrayValAllocate(). The saNtfPtrValAllocate() function uses the ptrVal field in SaNtfValueT, and the saNtfArrayValAllocate() function uses the arrayVal field in SaNtfValueT 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 to access the memory directly; instead, the application is supposed to access memory only by using the data pointers returned from the allocation functions (saNtfPtrValAllocate() or saNtfArrayValAllocate()) or the related get functions (saNtfPtrValGet() or saNtfArrayValGet()).

3.14.16 Additional Text

The type SaStringT is used for the additional text. A string consists of UTF-8 encoded characters and is terminated by the '\0' character.

3.14.17 SaNtfAdditionalInfoT

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

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

3.14.18 SaNtfProbableCauseT

This is the enumeration of probable causes as described in X.733 ([12]) and X.736 ([13]).

```
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;
```

3.14.19 SaNtfSpecificProblemT

```
typedef struct {
        SaNtfElementIdT problemId;
        /* API user is expected to define this field*/
        SaNtfClassIdT problemClassId;
        /* optional field to identify problemId values        *
         * from other notification class identifiers, needed  *
         * for correlation between clear and non-clear alarms *
         */
        SaNtfValueTypeT problemType;
        SaNtfValueT problemValue;
} SaNtfSpecificProblemT;
```

This type 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 notification class identifier of the notification. If it is specified, the given `problemId` value is taken from the notification class identifier given by `problemClassId`. If an alarm notification of perceived severity SA_NTF_SEVERITY_CLEARED contains a non-empty `specificProblems` parameter, the field `problemClassId` of each element in that parameter must be filled in to refer to the notification element identifier of the alarm that is to be cleared.

3.14.20 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;
```

3.14.21 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;
```

3.14.22 SaNtfThresholdInformationT

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

This type contains information about thresholds.

The fields `thresholdValue`, `thresholdHysteresis`, and `armTime` are intended to be used as specified in X.733 (see [12]).

The `thresholdValue`, `thresholdHysteresis`, and `observedValue` have to be of the same data type (as defined by the `thresholdValueType` field).

3.14.23 SaNtfProposedRepairActionT

```
typedef struct {
     SaNtfElementIdT actionId;
     /* API user is expected to define this field*/
     SaNtfValueTypeT actionValueType;
     SaNtfValueT actionValue;
} SaNtfProposedRepairActionT;
```

This type represents 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.

3.14.24 SaNtfSourceIndicatorT

```
typedef enum {
     SA_NTF_OBJECT_OPERATION          = 1,
     SA_NTF_MANAGEMENT_OPERATION      = 2,
     SA_NTF_UNKNOWN_OPERATION         = 3
} SaNtfSourceIndicatorT;
```

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

3.14.25 SaNtfStateChangeT_3

```
typedef struct {
     SaNtfElementIdT stateId;
     SaBoolT oldStatePresent;
     SaUint64T oldState;
     SaUint64T newState;
} SaNtfStateChangeT_3;
```

This type is used to represent state changes as part of a notification. The `oldState` and `newState` fields contain the old and the new state values, and the `stateId` field identifies the kind of state that has changed. The values of `stateId` are defined in the scope of a notification class identifier. The value of the optional field `oldState` is relevant only when `oldStatePresent` is SA_TRUE.

3.14.26 SaNtfAttributeT

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

This type is used to represent object attributes in an object creation or deletion notification.

3.14.27 SaNtfAttributeChangeT

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

This type is used to represent attribute changes in a notification. The values of `attributeId` are defined in the scope of a notification class identifier. The value of the optional field `oldAttributeValue` is relevant only when `oldAttributePresent` is SA_TRUE.

3.14.28 SaNtfServiceUserT

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

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

3.14.29 SaNtfSecurityAlarmDetectorT

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

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

3.14.30 SaNtfNotificationHeaderT

This type 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;
```

3.14.31 SaNtfObjectCreateDeleteNotificationT

This type 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;
```

3.14.32 SaNtfAttributeChangeNotificationT

This type 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;
```

3.14.33 SaNtfStateChangeNotificationT_3

This type 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_3 *changedStates;
     /* Points to changed states array in the internal*
      * notification structure*/
} SaNtfStateChangeNotificationT_3;
```

3.14.34 SaNtfAlarmNotificationT

This type 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;
```

3.14.35 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;
```

3.14.36 SaNtfMiscellaneousNotificationT

This type contains pointers to the fields of a miscellaneous notification.

```
typedef struct {
     SaNtfNotificationHandleT notificationHandle;
     /* A handle to the internal notification structure*/
     SaNtfNotificationHeaderT notificationHeader;
     /* Notification header*/
} SaNtfMiscellaneousNotificationT;
```

3.14.37 Default Variable Notification Data Size

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

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

3.14.38 SaNtfSubscriptionIdT

```
typedef SaUint32T SaNtfSubscriptionIdT;
```

This type is used for an identifier representing 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.

3.14.39 SaNtfNotificationFilterHeaderT

This type 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;
```

3.14.40 SaNtfObjectCreateDeleteNotificationFilterT

This type 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;
```

3.14.41 SaNtfAttributeChangeNotificationFilterT

This type 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;
```

3.14.42 SaNtfStateChangeNotificationFilterT_2

This type 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 */
     SaNtfElementIdT *stateId;
     /* the array of stateId values */
} SaNtfStateChangeNotificationFilterT_2;
```

Specifying one or more state id values (that is, when the array referred to by `stateId` has more than one element) makes only sense if there is also one notification class identifier specified in the structure designated by `notificationFilterHeader` (array of type `SaNtfClassIdT` pointed to by `notificationClassIds`). The state id values are related to that notification class identifier. More than one notification class identifier are also possible, but this makes sense only if they all define the same state ids.

3.14.43 SaNtfAlarmNotificationFilterT

This type 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;
```

3.14.44 SaNtfSecurityAlarmNotificationFilterT

This type 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;
```

3.14.45 SaNtfMiscellaneousNotificationFilterT

This type contains filter elements for a miscellaneous notification filter.

```
typedef struct {
     SaNtfNotificationFilterHandleT notificationFilterHandle;
     /* a handle to the internal notification filter structure */
     SaNtfNotificationFilterHeaderT notificationFilterHeader;
     /* the notification filter header */
} SaNtfMiscellaneousNotificationFilterT;
```

3.14.46 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.

3.14.47 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 type contains the search criteria for the Reader API.

3.14.48 SaNtfSearchDirectionT

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

This enumeration defines the search directions for the Reader API.

3.14.49 SaNtfNotificationTypeFilterHandlesT_3

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

```
typedef struct {
     SaNtfNotificationFilterHandleT objectCreateDeleteFilterHandle;
     SaNtfNotificationFilterHandleT attributeChangeFilterHandle;
     SaNtfNotificationFilterHandleT stateChangeFilterHandle;
     SaNtfNotificationFilterHandleT alarmFilterHandle;
     SaNtfNotificationFilterHandleT securityAlarmFilterHandle;
     SaNtfNotificationFilterHandleT miscellaneousFilterHandle;

} SaNtfNotificationTypeFilterHandlesT_3;
```

Unused handles in `SaNtfNotificationTypeFilterHandlesT_3` shall be set to `SA_NTF_FILTER_HANDLE_NULL`.

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

3.14.50 SaNtfNotificationsT_3

```
typedef struct {
     SaNtfNotificationTypeT notificationType;
     union
     {
          SaNtfObjectCreateDeleteNotificationT
               objectCreateDeleteNotification;
          SaNtfAttributeChangeNotificationT
               attributeChangeNotification;
          SaNtfStateChangeNotificationT_3
               stateChangeNotification;
          SaNtfAlarmNotificationT
               alarmNotification;
          SaNtfSecurityAlarmNotificationT
               securityAlarmNotification;
          SaNtfMiscellaneousNotificationT
               miscellaneousNotification;
     } notification;
} SaNtfNotificationsT_3;
```

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

3.14.51 SaNtfStateT

This enumeration lists the state identifiers for all states defined for the notification service.

```
typedef enum {
     SA_NTF_STATIC_FILTER_STATE       = 1,
     SA_NTF_SUBSCRIBER_STATE          = 2
} SaNtfStateT;
```

3.14.52 SaNtfStaticFilterStateT

This enumeration lists the state values for the state of a filter for static suppression of notifications (see SA_NTF_STATIC_FILTER_STATE in the preceding Section 3.14.51).

```
typedef enum {
     SA_NTF_STATIC_FILTER_STATE_INACTIVE    = 1,
     SA_NTF_STATIC_FILTER_STATE_ACTIVE      = 2
} SaNtfStaticFilterStateT;
```

3.14.53 SaNtfSubscriberStateT

This enumeration lists the state values for the state of a notification subscriber (see SA_NTF_SUBSCRIBER_STATE in Section 3.14.51).

```
typedef enum {
     SA_NTF_SUBSCRIBER_STATE_FORWARD_NOT_OK          = 1,
     SA_NTF_SUBSCRIBER_STATE_FORWARD_OK              = 2
} SaNtfSubscriberStateT;
```

3.14.54 Limit Enumeration

The Notification Service has no enumeration containing values that identify limits for a specific implementation of this service at the time of publication of this specification.

3.14.55 SaNtfNotificationMinorIdT

This type provides the values for the `minorId` field of notification class identifiers that the Notification Service uses in its own notifications. These notifications are described in Section 6.2 on page 166.

```
typedef enum {
     /* State Change */
     SA_NTF_NTFID_STATIC_FILTER_ACTIVATED       = 0x065,
     SA_NTF_NTFID_STATIC_FILTER_DEACTIVATED     = 0x066,
     SA_NTF_NTFID_CONSUMER_SLOW                 = 0x067,
     SA_NTF_NTFID_CONSUMER_FAST_ENOUGH          = 0x068
} SaNtfNotificationMinorIdT;
```

3.15 Library Life Cycle

3.15.1 saNtfInitialize_3()

Prototype

```
SaAisErrorT saNtfInitialize_3(
      SaNtfHandleT *ntfHandle,
      const SaNtfCallbacksT_3 *ntfCallbacks,
      SaVersionT *version
);
```

Parameters

`ntfHandle` - [`out`] A pointer to the handle which designates this particular initialization of the Notification Service and which is to be returned by the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`ntfCallbacks` - [`in`] If `ntfCallbacks` is set to NULL, no callback is registered; if `ntfCallbacks` is not set to NULL, it is a pointer to an `SaNtfCallbacksT_3` structure that contains the callback functions of the process that the Notification Service may invoke. Only non-NULL callback functions in this structure will be registered. The type `SaNtfCallbacksT_3` is defined in Section 3.14.2 on page 49.

`version` - [`in/out`] As an input parameter, `version` is a pointer to a structure containing the required Notification Service version. In this case, `minorVersion` is ignored and should be set to 0x00. As an output parameter, `version` is a pointer to a structure containing the version actually supported by the Notification Service. The `SaVersionT` type is defined in [2].

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 pointed to by `ntfHandle` is returned by the Notification Service 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 version of the Notification Service API specified by the `releaseCode` and `majorVersion` fields of the structure pointed to by the `version` parameter, `SA_AIS_OK` is returned. In this case, the structure pointed to by the `version` parameter is set by this function to:

- `releaseCode` = required release code
- `majorVersion` = highest value of the major version that this implementation cansupport 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 preceding condition cannot be met, `SA_AIS_ERR_VERSION` is returned, and the structure pointed to by 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 lowest 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 lower 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 provided in the structure to which the `version` parameter points is not compatible with the version of the Notification Service implementation.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node because it is not a member node.

See Also

`saNtfSelectionObjectGet()`, `saNtfDispatch()`, `saNtfFinalize()`

3.15.2 saNtfSelectionObjectGet()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`selectionObject` - [out] A pointer to the operating system handle that the invoking process can use to detect pending callbacks. The `SaSelectionObjectT` type is defined in [2].

Description

This function returns the operating system handle associated with the handle `ntfHandle`. The invoking process can use the operating system 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 operating system handle 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).

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfDispatch()`, `saNtfFinalize()`

3.15.3 saNtfDispatch()

Prototype

```
SaAisErrorT saNtfDispatch(
      SaNtfHandleT ntfHandle,
      SaDispatchFlagsT dispatchFlags
);
```

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`dispatchFlags` - [in] Flags that specify the callback execution behavior of the `saNtfDispatch()` function. These flags have the values `SA_DISPATCH_ONE`, `SA_DISPATCH_ALL`, or `SA_DISPATCH_BLOCKING`. These flags are values of the `SaDispatchFlagsT` enumeration type, which is described in [2].

Description

In the context of the calling thread, this function invokes 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. This value is also returned if this function is being invoked with `dispatchFlags` set to `SA_DISPATCH_ALL` or `SA_DISPATCH_BLOCKING`, and the handle `ntfHandle` has been finalized.

`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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle ntfHandle was acquired before the cluster node left the cluster membership.

See Also

saNtfInitialize_3(), saNtfFinalize()

3.15.4 saNtfFinalize()

Prototype

```
SaAisErrorT saNtfFinalize(
     SaNtfHandleT ntfHandle
);
```

Parameters

ntfHandle - [in] The handle which was obtained by a previous invocation of saNtfInitialize_3() and which designates this particular initialization of the Notification Service. The SaNtfHandleT type is defined in Section 3.14.1.1 on page 48.

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_3() before it invokes this function. A process must invoke this function once for each handle it acquired by invoking saNtfInitialize_3().

If the saNtfFinalize() function completes successfully, it releases all resources acquired when saNtfInitialize_3() was called; it uninstalls any subscriptions of this process to receive notifications (which were installed by saNtfNotificationSubscribe_3()) and frees any resources allocated by the Notification Service for the subscription; additionally, it finalizes any reading of logged notifications (which were initialized by saNtfNotificationReadInitialize_3()) and frees any resources allocated by the Notification Service for the reading. Moreover, it frees any notifications allocated in the SaNtfNotificationCallbackT_3 and the saNtf<notification type>NotificationAllocate() functions (which have

not yet been freed by `saNtfNotificationFree()`), any notification filters allocated in the `saNtf<notification type>NotificationFilterAllocate()` functions (which have not yet been freed by `saNtfNotificationFilterFree()`), and any localized messages allocated in the `saNtfLocalizedMessageGet()` function that have not yet been freed by `saNtfLocalizedMessageFree_2()`. Furthermore, `saNtfFinalize()` 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.

If a process terminates, the Notification Service implicitly finalizes all instances of the Notification Service that are associated with the process, as described in the preceding paragraph.

After `saNtfFinalize()` completes successfully, the handle `ntfHandle` and the selection object associated with it are 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()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationUnsubscribe_2()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadFinalize()`, `saNtfLocalizedMessageGet()`, `saNtfLocalizedMessageFree_2()`, `saNtfObjectCreateDeleteNotificationAllocate()`, `saNtfAttributeChangeNotificationAllocate()`, `saNtfStateChangeNotificationAllocate_3()`, `saNtfAlarmNotificationAllocate()`, `saNtfSecurityAlarmNotificationAllocate()`, `saNtfMiscellaneousNotificationAllocate()`, `saNtfNotificationFree()`, `saNtfObjectCreateDeleteNotificationFilterAllocate()`,

`saNtfAttributeChangeNotificationFilterAllocate()`,
`saNtfStateChangeNotificationFilterAllocate_2()`,
`saNtfAlarmNotificationFilterAllocate()`,
`saNtfSecurityAlarmNotificationFilterAllocate()`,
`saNtfMiscellaneousNotificationFilterAllocate()`,
`saNtfNotificationFilterFree()`, `SaNtfNotificationCallbackT_3`,
`SaNtfNotificationDiscardedCallbackT`,
`SaNtfStaticSuppressionFilterSetCallbackT_3`,
`saNtfSelectionObjectGet()`

3.16 Operations of the Producer API

This section describes the API functions that enable the caller to generate notifications.

Generation of notifications is typically divided into six steps:

1. Optionally specifying an `SaNtfStaticSuppressionFilterSetCallbackT_3` callback when calling `saNtfInitialize_3()` to obtain information about notification types that are currently entirely statically suppressed and, therefore, make no sense to be generated.
2. Allocating memory for the notification contents with one of the allocation functions described in subsections of this section.
3. Optionally invoking `saNtfVariableDataSizeGet()` to determine the memory space available.
4. Filling in the notification fields of the structure allocated in step 2.
5. Calling the function `saNtfNotificationSend()` with the notification handle returned in step 2.
6. Releasing the allocated memory with the `saNtfNotificationFree()` function.

Steps 4. and 5. 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 lower, but must not be higher 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 that was allocated with `saNtfPtrValAllocate()` or `saNtfArrayValAllocate()` may be less, but must not be greater than the size that was specified with the allocate function.

In some situations, a notification producer may need to complete the processing triggered by a particular event before it can generate the corresponding notification that reports the event. Thus, it may be convenient for the producer to use the identifier of this notification that it will send later as an identifier for the event being processed. In this case, the producer can invoke the `saNtfIdentifierAllocate()` function at anytime before step 5. to preallocate a notification identifier. Subsequently, in step 5., `saNtfNotificationSendWithId()` can be used instead of `saNtfNotificationSend()` to send the notification with the preallocated notification identifier.

3.16.1 saNtfObjectCreateDeleteNotificationAllocate()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notification` - [out] A pointer to a structure of `SaNtfObjectCreateDeleteNotificationT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfObjectCreateDeleteNotificationT` type is defined in Section 3.14.31 on page 65.

`numCorrelatedNotifications` - [in] Number of correlated notifications in the notification. The `SaUint16T` type is defined in [2].

`lengthAdditionalText` - [in] Length of additional text in bytes (including terminating '\0'). The `SaUint16T` type is defined in [2].

`numAdditionalInfo` - [in] Number of additional info fields. The `SaUint16T` type is defined in [2].

`numAttributes` - [in] Number of object attributes in the notification. The `SaUint16T` type is defined in [2].

`variableDataSize` - [in] The maximum number of bytes that are used 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 to get PDU-ready notifications. The system limit of the Notification Service is allocated if `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified. If `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified, and the notification was successfully allocated, the `saNtfVariableDataSizeGet()` function can be invoked to determine the maximum size available. The `SaUint16T` type is defined in [2].

Description

This API internally allocates memory for an object create/delete notification and initializes the structure pointed to by the `notification` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notification` parameter. The pointers in the structure referred to by the `notification` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notification` parameter also contains the notification handle, which is used for subsequent calls to functions such as `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, 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 `ntfHandle` 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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, `saNtfNotificationFree()`, `saNtfVariableDataSizeGet()`, `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`

3.16.2 saNtfAttributeChangeNotificationAllocate()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notification` - [out] A pointer to a structure of `SaNtfAttributeChangeNotificationT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfAttributeChangeNotificationT` type is defined in Section 3.14.32 on page 66.

`numCorrelatedNotifications` - [in] Number of correlated notifications in the notification. The `SaUint16T` type is defined in [2].

`lengthAdditionalText` - [in] Length of additional text in bytes (including terminating '\0'). The `SaUint16T` type is defined in [2].

`numAdditionalInfo` - [in] Number of additional info fields. The `SaUint16T` type is defined in [2].

`numAttributes` - [in] Number of changed attributes in the notification. The `SaUint16T` type is defined in [2].

`variableDataSize` - [in] The maximum number of bytes that are used 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 to get PDU-ready notifications. The system limit of the Notification Service is allocated if `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified. If `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified, and the notification was successfully allocated, the `saNtfVariableDataSizeGet()` function can be invoked to determine the maximum size available. The `SaUint16T` type is defined in [2].

Description

This API internally allocates memory for an attribute change notification and initializes the structure pointed to by the `notification` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notification` parameter. The pointers in the structure referred to by the `notification` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notification` parameter also contains the notification handle, which is used for subsequent calls to functions such as `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, 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 `ntfHandle` 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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, `saNtfNotificationFree()`, `saNtfVariableDataSizeGet()`, `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`

3.16.3 saNtfStateChangeNotificationAllocate_3()

Prototype

```
SaAisErrorT saNtfStateChangeNotificationAllocate_3(
      SaNtfHandleT ntfHandle,
      SaNtfStateChangeNotificationT_3 *notification,
      SaUint16T numCorrelatedNotifications,
      SaUint16T lengthAdditionalText,
      SaUint16T numAdditionalInfo,
      SaUint16T numStateChanges,
      SaInt16T variableDataSize
);
```

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notification` - [`out`] A pointer to a structure of `SaNtfStateChangeNotificationT_3` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfStateChangeNotificationT_3` type is defined in Section 3.14.33 on page 66.

`numCorrelatedNotifications` - [`in`] Number of correlated notifications in the notification. The `SaUint16T` type is defined in [2].

`lengthAdditionalText` - [`in`] Length of additional text in bytes (including terminating '\0'). The `SaUint16T` type is defined in [2].

`numAdditionalInfo` - [`in`] Number of additional info fields. The `SaUint16T` type is defined in [2].

`numStateChanges` - [`in`] Number of changed states in the notification. The `SaUint16T` type is defined in [2].

`variableDataSize` - [`in`] The maximum number of bytes that are used 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 to get PDU-ready notifications. The system limit of the Notification Service is allocated if `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified. If `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified, and the notification was successfully allocated, the `saNtfVariableDataSizeGet()` function can be invoked to determine the maximum size available. The `SaUint16T` type is defined in [2].

Description

This API internally allocates memory for a state change notification and initializes the structure pointed to by the `notification` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notification` parameter. The pointers in the structure referred to by the `notification` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notification` parameter also contains the notification handle, which is used for subsequent calls to functions such as `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, 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 `ntfHandle` 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.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, `saNtfNotificationFree()`, `saNtfVariableDataSizeGet()`, `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`

3.16.4 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 which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notification` - [out] A pointer to a structure of `SaNtfAlarmNotificationT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfAlarmNotificationT` type is defined in Section 3.14.34 on page 67.

`numCorrelatedNotifications` - [in] Number of correlated notifications in the notification. The `SaUint16T` type is defined in [2].

`lengthAdditionalText` - [in] Length of additional text in bytes (including terminating '\0'). The `SaUint16T` type is defined in [2].

`numAdditionalInfo` - [in] Number of additional info fields. The `SaUint16T` type is defined in [2].

`numSpecificProblems` - [in] Number of specific problems. The `SaUint16T` type is defined in [2].

`numMonitoredAttributes` - [in] Number of monitored attributes. The `SaUint16T` type is defined in [2].

`numProposedRepairActions` - [in] Number of proposed repair actions. The `SaUint16T` type is defined in [2].

`variableDataSize` - [in] The maximum number of bytes that are used 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 to get PDU-ready notifications. The system limit of the Notification Service is allocated if `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified. If `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified, and the notification was successfully allocated, the `saNtfVariableDataSizeGet()` function can be invoked to determine the maximum size available.

Description

This API internally allocates memory for an alarm notification and initializes the structure pointed to by the `notification` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the related attributes in the structure referred to by the `notification` parameter. The pointers in the structure pointed to by the `notification` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notification` parameter also contains the notification handle, which is used for subsequent calls to functions such as `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, 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 `ntfHandle` 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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, `saNtfNotificationFree()`, `saNtfVariableDataSizeGet()`, `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`

3.16.5 saNtfSecurityAlarmNotificationAllocate()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notification` - [`out`] A pointer to a structure of `SaNtfSecurityAlarmNotificationT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfSecurityAlarmNotificationT` type is defined in Section 3.14.35 on page 68.

`numCorrelatedNotifications` - [`in`] Number of correlated notifications in the notification. The `SaUint16T` type is defined in [2].

`lengthAdditionalText` - [`in`] Length of additional text in bytes (including terminating '\0'). The `SaUint16T` type is defined in [2].

`numAdditionalInfo` - [`in`] Number of additional info fields. The `SaUint16T` type is defined in [2].

`variableDataSize` - [`in`] The maximum number of bytes that are used 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 to get PDU-ready notifications. The system limit of the Notification Service is allocated if `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified. If `SA_NTF_ALLOC_SYSTEM_LIMIT` is specified, and the notification was successfully allocated, the `saNtfVariableDataSizeGet()` function can be invoked to determine the maximum size available. The `SaUint16T` type is defined in [2].

Description

This API internally allocates memory for a security alarm notification and initializes the structure pointed to by the `notification` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notification` parameter. The pointers in the structure referred to by the `notification` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notification` parameter also contains the notification handle, which is used for subsequent calls to functions such as `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, 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 `ntfHandle` 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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSend()`, `saNtfNotificationSendWithId()`, `saNtfNotificationFree()`, `saNtfVariableDataSizeGet()`, `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`

3.16.6 saNtfMiscellaneousNotificationAllocate()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notification` - [out] A pointer to a structure of `SaNtfMiscellaneousNotificationT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfMiscellaneousNotificationT` type is defined in Section 3.14.36 on page 68.

`numCorrelatedNotifications` - [in] Number of correlated notifications in the notification. The `SaUint16T` type is defined in [2].

`lengthAdditionalText` - [in] Length of additional text in bytes (including terminating '\0'). The `SaUint16T` type is defined in [2].

`numAdditionalInfo` - [in] Number of additional info fields. The `SaUint16T` type is defined in [2].

`variableDataSize` - [in] The maximum number of bytes that are used 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 to get PDU-ready notifications. The system limit of the Notification Service is allo-

cated if SA_NTF_ALLOC_SYSTEM_LIMIT is specified. If SA_NTF_ALLOC_SYSTEM_LIMIT is specified, and the notification was successfully allocated, the saNtfVariableDataSizeGet() function can be invoked to determine the maximum size available. The SaUint16T type is defined in [2].

Description

This API internally allocates memory for a miscellaneous notification and initializes the structure pointed to by the notification parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the notification parameter. The pointers in the structure referred to by the notification parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the notification parameter also contains the notification handle, which is used for subsequent calls to functions such as saNtfPtrValAllocate(), saNtfArrayValAllocate(), saNtfNotificationSend(), saNtfNotificationSendWithId(), 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 ntfHandle 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.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle ntfHandle was acquired before the cluster node left the cluster membership.

See Also

saNtfInitialize_3(), saNtfNotificationSend(), saNtfNotificationSendWithId(), saNtfNotificationFree(), saNtfVariableDataSizeGet(), saNtfPtrValAllocate(), saNtfArrayValAllocate()

3.16.7 saNtfPtrValAllocate()

Prototype

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

Parameters

notificationHandle - [in] The handle which was obtained by a previous call to one of the saNtf<notification type>NotificationAllocate() functions and which identifies the particular notification instance for which memory is to be reserved. The SaNtfNotificationHandleT type is defined in Section 3.14.1.2 on page 48.

dataSize - [in] The number of bytes to be reserved. The SaUint16T type is defined in [2].

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

value - [in/out] A pointer to an element of the notification structure that is passed in, and 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 when the `saNtfPtrValAllocate()` function is called. Memory allocated by this function is implicitly freed when `saNtfNotificationFree()` is called. The offset and length of the reserved memory space is stored in the element pointed to by `value`. The `SaNtfValueT` type is defined in Section 3.14.15 on page 55.

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 the field pointed to by `dataPtr`. This function may only be used with the `ptrVal` structure field in the `SaNtfValueT` union, that is, for the following 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 not enough space is left.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;

- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationAllocate()`, `saNtfAttributeChangeNotificationAllocate()`, `saNtfStateChangeNotificationAllocate_3()`, `saNtfAlarmNotificationAllocate()`, `saNtfSecurityAlarmNotificationAllocate()`, `saNtfMiscellaneousNotificationAllocate()`, `saNtfPtrValGet()`, `saNtfNotificationFree()`

3.16.8 saNtfArrayValAllocate()

Prototype

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

Parameters

`notificationHandle` - [in] The handle which was obtained by a previous call to one of the `saNtf<notification type>NotificationAllocate()` functions and which identifies the particular notification instance for which memory is to be reserved. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`numElements` - [in] Number of elements to be reserved. The `SaUint16T` type is defined in [2].

`elementSize` - [in] Size of each element in the array in bytes. The `SaUint16T` type is defined in [2].

`arrayPtr` - [out] A pointer to a pointer to the memory location that will be reserved with this function.

`value` - [`in/out`] A pointer to an element of the notification structure that is passed in, and 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 when the `saNtfArrayValAllocate()` function is called. Memory allocated by 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 the element pointed to by `value`. The `SaNtfValueT` type is defined in Section 3.14.15 on page 55.

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 the field pointed to by `arrayPtr`. This function may only be used with the `arrayVal` structure field in the `SaNtfValueT` union, that is, 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 not enough space is left.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationAllocate()`, `saNtfAttributeChangeNotificationAllocate()`, `saNtfStateChangeNotificationAllocate_3()`, `saNtfAlarmNotificationAllocate()`, `saNtfSecurityAlarmNotificationAllocate()`, `saNtfMiscellaneousNotificationAllocate()`, `saNtfArrayValGet()`, `saNtfNotificationFree()`

3.16.9 saNtfIdentifierAllocate()

Prototype

```
SaAisErrorT saNtfIdentifierAllocate(
      SaNtfNotificationHandleT notificationHandle,
      SaNtfIdentifierT *notificationIdentifier
);
```

Parameters

`notificationHandle` - [in] The handle which was obtained by a previous call to one of the `saNtf<notification type>NotificationAllocate()` functions and which identifies the particular notification instance for which memory is to be reserved. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`notificationIdentifier` - [out] A pointer to the notification identifier that the Notification Service generates. This notification identifier can be used later when sending a notification by invoking the `saNtfNotificationSendWithId()` function. The `SaNtfIdentifierT` type is defined in Section 3.14.11 on page 53.

Description

This function allows users of the Notification Service to preallocate a notification identifier for a notification they will send later on.

The usage of this function shall be limited to users of the Notification Service that execute under realtime constraints and need to defer sending a notification to a later time but need to communicate the notification identifier of this future notification to other users that will use the notification identifier as a correlation identifier in the notifications they generate.

The returned identifier shall only be used once to send 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.

`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_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfNotificationSend()`, `saNtfNotificationSendWithId()`

3.16.10 saNtfNotificationSend() and saNtfNotificationSendWithId()

Prototype

```
SaAisErrorT saNtfNotificationSend(
      SaNtfNotificationHandleT notificationHandle
);

SaAisErrorT saNtfNotificationSendWithId(
      SaNtfNotificationHandleT notificationHandle,
      SaNtfIdentifierT notificationIdentifier
);
```

Parameters

`notificationHandle` - [in] The handle which was obtained by a previous call to one of the `saNtf<notification type>NotificationAllocate()` functions and which designates this particular notification instance. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`notificationIdentifier` - [in] The notification identifier to be assigned to the notification to be sent. This notification identifier must have been previously allocated by invoking the `saNtfIdentifierAllocate()` function, and it must not already have been used previously to send a notification successfully. The `SaNtfIdentifierT` type is defined in Section 3.14.11 on page 53.

Description

These functions are 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 referred to by `notificationHandle` are mandatory, optional, or implicit:

Table 9 Properties of Header Elements

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[1]	-
`correlatedNotifications`	optional – number of elements must be consistent with numCorrelatedNotifications	–
`additionalText`	optional – length must be consistent with lengthAdditionalText	–
`additionalInfo`	optional – number of elements must be consistent with `numAdditionalInfo`	–

1. Except for `saNtfNotificationSendWithId()`, in which case `notificationId` must be specified in a parameter.

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

Table 10 Properties of Elements in Object Create/Delete Notifications

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
`notificationHandle`	implicit (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:

Table 11 Properties of Elements in Attribute Value Change Notifications

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
`notificationHandle`	implicit (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:

Table 12 Properties of Elements in State Change Notifications

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
`notificationHandle`	implicit (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:

Table 13 Properties of Elements in Alarms

Name	Mandatory / Optional / Implicit	Default Value for Optional Parameters (if not set)
`notificationHandle`	implicit (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:

Table 14 Properties of Elements in Security Alarms

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

In case `notificationHandle` refers to a **miscellaneous notification**, the `notificationHandle` is implicit and there are no optional related elements defined.

If `saNtfNotificationSend()` is invoked to send the notification, the notification identifier is allocated by the Notification Service during the call; however, if `saNtfNotificationSendWithId()` is invoked, the preallocated identifier provided by the caller in `notificationIdentifier` is used as the notification identifier.

In both cases, if the notification is sent successfully, the notification identifier is written to the field pointed to by `notificationId` in the `SaNtfNotificationHeaderT` part of the notification identified by `notificationHandle`. The notification identifier can later be used to refer to this notification, for example, as a correlation identifier included in other notifications.

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_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library. This return value applies only to the `saNtfNotificationSendWithId()` function.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;

- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationAllocate()`,
`saNtfAttributeChangeNotificationAllocate()`,
`saNtfStateChangeNotificationAllocate_3()`,
`saNtfAlarmNotificationAllocate()`,
`saNtfSecurityAlarmNotificationAllocate()`,
`saNtfMiscellaneousNotificationAllocate()`

3.16.11 saNtfNotificationFree()

Prototype

```
SaAisErrorT saNtfNotificationFree(
      SaNtfNotificationHandleT notificationHandle
);
```

Parameters

`notificationHandle` - [in] The handle which was obtained by a previous call to one of the `saNtf<notification type>NotificationAllocate()` functions and which designates this particular notification instance. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle notificationHandle was acquired before the cluster node left the cluster membership.

See Also

saNtfObjectCreateDeleteNotificationAllocate(), saNtfAttributeChangeNotificationAllocate(), saNtfStateChangeNotificationAllocate_3(), saNtfAlarmNotificationAllocate(), saNtfSecurityAlarmNotificationAllocate(), saNtfMiscellaneousNotificationAllocate(), saNtfNotificationSend(), saNtfNotificationSendWithId(), SaNtfNotificationCallbackT_3

3.16.12 SaNtfStaticSuppressionFilterSetCallbackT_3

Prototype

```
typedef void (*SaNtfStaticSuppressionFilterSetCallbackT_3)(
     SaNtfHandleT ntfHandle,
     SaNtfEventTypeBitmapT eventTypeBitmap
);
```

Parameters

ntfHandle - [in] The handle which was obtained by a previous invocation of saNtfInitialize_3() and which designates this particular initialization of the Notification Service. The SaNtfHandleT type is defined in Section 3.14.1.1 on page 48.

eventTypeBitmap - [in] This bitmap has the bits set (bits have the value 1) for those event types that are currently completely suppressed. Any combination of the bits defined for the SaNtfEventTypeBitmapT type in Section 3.14.5 on page 51 may be set. Other bits have no meaning and must always be set to 0. If one of the defined bits is set, all notifications of the event type related to the bit are statically

suppressed. Note that the corresponding bit for an event type is set if and only if there is an active filter matching all notifications of that event type. If no filter is active, or the active filters match only some notifications of that event type, the corresponding bit is not set (the bit has the value 0).

Description

The Notification Service invokes this callback to inform the process about the event types for which all notifications are statically suppressed in `eventTypeBitmap`. Ideally, a Notification Producer should not produce notifications for those event types. Especially during overload situations of the system, the close-to-source suppression of notifications is meaningful. If, however, a Notification Producer does produce such notifications, or it did not provide this callback function, the Notification Service shall discard the notification. In other words, this callback allows for close-to-source suppression of notifications if the Notification Producers provide it and respect the information in `eventTypeBitmap`.
Regardless of the behavior of Notification Producers, the Notification Service must suppress notifications when one or more filters are active.

This callback is invoked in the context of a thread calling `saNtfDispatch()` with the handle `ntfHandle` that was specified when the Notification Service was initialized by an invocation of `saNtfInitialize_3()`.

This callback may be invoked for one of the two reasons.

- A call to `saNtfInitialize_3()` has been executed successfully. The process is initially informed about the current status of the static suppression.
- The value passed to the callback in `eventTypeBitmap` has changed since the last time the callback was invoked, that is, at least one of the bits that were not set before are now set or vice versa. The bits change their values as a result of either an administrative operation to activate or deactivate a static suppression filter or after the configuration data of an active static suppression filter has been modified.

Return Values

None

See Also

`saNtfInitialize_3()`, `saNtfDispatch()`

3.16.13 saNtfVariableDataSizeGet()

Prototype

```
SaAisErrorT saNtfVariableDataSizeGet(
      SaNtfNotificationHandleT notificationHandle,
      SaUint16T *variableDataSpaceAvailable
);
```

Parameters

`notificationHandle` - [in] The handle which was obtained by a previous call to one of the `saNtf<notification type>NotificationAllocate()` functions and which designates this particular notification instance for which the maximum data size available for variable data parts is to be determined. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`variableDataSpaceAvailable` - [out] A pointer to the variable data size value in bytes. This amount of data space is available for the variable data parts of one or more `SaNtfValueT` elements used in the notification. (`SaNtfValueT` elements with one of these value types have data parts of variable length: SA_NTF_VALUE_LDAP_NAME, SA_NTF_VALUE_STRING, SA_NTF_VALUE_IPADDRESS, SA_NTF_VALUE_BINARY, or SA_NTF_VALUE_ARRAY.) Memory space for the variable data parts of the `SaNtfValueT` elements can be allocated by invoking `saNtfPtrValAllocate()` or `saNtfArrayValAllocate()`. The `SaUint16T` type is defined in [2].

Description

This function returns in the field pointed to by `variableDataSpaceAvailable` the amount of memory space available for data elements of variable size in the context of the notification referred to by `notificationHandle`. The total space available for these data elements depends on the `variableDataSize` parameter passed to the `saNtf<notification type>Allocate()` function and on the limits given by an implementation of the Notification Service. When `saNtfVariableDataSizeGet()` is called after the `saNtf<notification type>Allocate()` function and before `saNtfPtrValAllocate()` or `saNtfArrayValAllocate()`, the value returned in the field pointed to by `variableDataSpaceAvailable` is the same value which was passed as `variableDataSize` parameter to the `saNtf<notification type>Allocate()` function, unless SA_NTF_ALLOC_SYSTEM_LIMIT was specified as `variableDataSize`. In the latter case, the value pointed to by `variableDataSpaceAvailable` is the maximum

available size. The maximum size depends on the implementation of the Notification Service (maximum size of a PDU) and may also depend on the memory space used for other variable parts of the notification, for example, for the number of additional information elements. The `saNtfVariableDataSizeGet()` function may be called after one or more calls to `saNtfPtrValAllocate()` or `saNtfArrayValAllocate()`. In such a case, the value returned in the field pointed to by `variableDataSpaceAvailable` is the remaining space available for additional calls to `saNtfPtrValAllocate()` or `saNtfArrayValAllocate()`.

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_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationAllocate()`, `saNtfAttributeChangeNotificationAllocate()`, `saNtfStateChangeNotificationAllocate_3()`, `saNtfAlarmNotificationAllocate()`, `saNtfSecurityAlarmNotificationAllocate()`, `saNtfMiscellaneousNotificationAllocate()`, `saNtfPtrValAllocate()`, `saNtfArrayValAllocate()`

3.17 Consumer Operations

3.17.1 Common Consumer Operations

This section contains functions that are common to the Subscriber and Reader API, namely 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.

3.17.1.1 saNtfLocalizedMessageGet()

Prototype

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

Parameters

`notificationHandle` - [`in`] notification handle. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`message` - [`out`] A pointer to a pointer to the buffer to contain the returned localized message. The `message` pointer must not be NULL. If the function returns successfully, `message` points to a pointer which, in turn, points to memory allocated by the function and which contains the localized message. The calling application is responsible for freeing this memory when it is no longer needed by invoking `saNtfLocalizedMessageFree_2()`. The function `saNtfLocalizedMessageFree_2()` can be invoked before or after the notification referred to by `notificationHandle` has been freed by calling `saNtfNotificationFree()`.
If `saNtfLocalizedMessageGet()` returns an error, no memory has been allocated for the buffer for the localized messages. The `SaStringT` type is defined in [2].

Description

This function returns a localized textual description of the situation that resulted in the notification referred to 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_3` callback or the

`saNtfNotificationReadNext_3()` function. If no localization data is available for the notification class identifier in the notification referred to by `notificationHandle`, this function returns `SA_AIS_ERR_NOT_EXIST`. Localization data are optional and need not be provided for all notification class identifiers. If 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, these references are usually replaced by the values referred to. If the value referred to is not contained in the notification, the returned message keeps the reference as it is in the format string. Refer to Appendix B on page 175 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` - No localization data is available for the notification class identifier in the notification that is referred to by `notificationHandle`.

`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_NOT_SUPPORTED` - The implementation of the Notification Service does not support the optional functionality of internationalization.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;

- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`SaNtfNotificationCallbackT_3`, `saNtfLocalizedMessageFree_2()`, `saNtfNotificationFree()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfInitialize_3()`, `saNtfNotificationReadNext_3()`

3.17.1.2 saNtfLocalizedMessageFree_2()

Prototype

```
SaAisErrorT saNtfLocalizedMessageFree_2(
     SaNtfHandleT ntfHandle,
     SaStringT message
);
```

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`message` - [in] A pointer to a pointer which in turn points to the message buffer that was obtained by a previous invocation of the `saNtfLocalizedMessageGet()` function. The `SaStringT` type is defined in [2].

Description

This function 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.

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

SA_AIS_ERR_INVALID_PARAM - A parameter is invalid.

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

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfLocalizedMessageGet()`

3.17.1.3 saNtfPtrValGet()

Prototype

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

Parameters

`notificationHandle` - [`in`] The notification handle that was obtained from the notification structure passed to one of the callbacks of the Subscriber API or returned from one of the functions of the Reader API. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`value` - [`in`] A pointer to an element of the notification structure whose data is to be returned. The offset and length of the reserved memory space is taken from this element. The `SaNtfValueT` type is defined in Section 3.14.15 on page 55.

`dataPtr` - [`out`] A pointer to a pointer to the returned data. Since the returned pointer points to an internal data structure, the returned pointer is no longer valid after the notification referred to by `notificationHandle` has been freed with `saNtfNotificationFree()`.

`dataSize` - [`out`] A pointer to the size of the data associated with the value pointed to by `value`. The `SaUint16T` type is defined in [2].

Description

This function obtains the pointer to a memory location, which was allocated in an internal structure associated with the notification instance to which `notificationHandle` refers. The `ptrVal` field in the structure pointed to by `value` is used by `saNtfPtrValGet()` to derive the pointer value (the pointer referred to by `dataPtr`) and the size of the data (returned in the field pointed to by `dataSize`). The calling process has to know the semantics of the returned data, that is, it has to know its structure or data type.

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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationAllocate()`, `saNtfAttributeChangeNotificationAllocate()`, `saNtfStateChangeNotificationAllocate_3()`, `saNtfAlarmNotificationAllocate()`, `saNtfSecurityAlarmNotificationAllocate()`, `saNtfMiscellaneousNotificationAllocate()`, `saNtfPtrValAllocate()`, `saNtfNotificationFree()`

3.17.1.4 saNtfArrayValGet()

Prototype

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

Parameters

`notificationHandle` - [`in`] The notification handle that was obtained from the notification structure passed to one of the callbacks of the Subscriber API or returned from one of the functions of the Reader API. The `SaNtfNotificationHandleT` type is defined in Section 3.14.1.2 on page 48.

`value` - [`in`] A pointer to an element of the notification structure whose data is to be returned. The offset, size, and field width of the reserved array space is taken from this element. The `SaNtfValueT` type is defined in Section 3.14.15 on page 55.

`arrayPtr` - [`out`] A pointer to a pointer to the returned array. Since the returned pointer points to an internal data structure, the returned pointer is no longer valid after the notification referred to by `notificationHandle` has been freed with `saNtfNotificationFree()`.

`numElements` - [`out`] A pointer to the number of elements in the array. The `SaUint16T` type is defined in [2].

`elementSize` - [`out`] A pointer to the size of each element in the array. The `SaUint16T` type is defined in [2].

Description

This function obtains the pointer to an array, which was allocated in an internal structure associated with the notification instance to which `notificationHandle` refers. The `arrayVal` field in the structure pointed to by `value` is used by `saNtfArrayValGet()` to derive the value of the pointer to the array (the pointer referred to by `arrayPtr`), the size of each element in the array (returned in the field pointed to by `elementSize`), and the number of elements in the array (returned in

the field pointed to by `numElements`). The calling process has to know the semantics of the returned data.

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.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `notificationHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationAllocate()`, `saNtfAttributeChangeNotificationAllocate()`, `saNtfStateChangeNotificationAllocate_3()`, `saNtfAlarmNotificationAllocate()`, `saNtfSecurityAlarmNotificationAllocate()`, `saNtfMiscellaneousNotificationAllocate()`, `saNtfArrayValAllocate()`, `saNtfNotificationFree()`

3.17.1.5 saNtfObjectCreateDeleteNotificationFilterAllocate()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notificationFilter` - [out] A pointer to a structure of `SaNtfObjectCreateDeleteNotificationFilterT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfObjectCreateDeleteNotificationFilterT` type is defined in Section 3.14.40 on page 70.

`numEventTypes` - [in] Number of event types in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationObjects` - [in] Number of notification objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotifyingObjects` - [in] Number of notifying objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationClassIds` - [in] Number of notification class ids in the notification filter. The `SaUint16T` type is defined in [2].

`numSourceIndicators` - [in] Number of source indicators in the notification filter. The `SaUint16T` type is defined in [2].

Description

This function internally allocates memory for an object create/delete notification filter and initializes the structure pointed to by the `notificationFilter` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notificationFilter` parameter. The pointers in the structure referred to by the `notificationFilter` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notificationFilter` parameter also contains the notification filter handle, which is used for subsequent calls to functions such as `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, 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 `ntfHandle` 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_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, `saNtfNotificationFilterFree()`

3.17.1.6 saNtfAttributeChangeNotificationFilterAllocate()

Prototype

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

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notificationFilter` - [out] A pointer to a structure of `SaNtfAttributeChangeNotificationFilterT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfAttributeChangeNotificationFilterT` type is defined in Section 3.14.41 on page 70.

`numEventTypes` - [in] Number of event types in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationObjects` - [in] Number of notification objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotifyingObjects` - [in] Number of notifying objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationClassIds` - [in] Number of notification class ids in the notification filter. The `SaUint16T` type is defined in [2].

`numSourceIndicators` - [in] Number of source indicators in the notification filter. The `SaUint16T` type is defined in [2].

Description

This function internally allocates memory for an attribute change notification filter and initializes the structure pointed to by the `notificationFilter` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notificationFilter` parameter. The pointers in the structure referred to by the `notificationFilter` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notificationFilter` parameter also contains the notification filter handle, which is used for subsequent calls to functions such as `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, 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 `ntfHandle` 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_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, `saNtfNotificationFilterFree()`

3.17.2 saNtfStateChangeNotificationFilterAllocate_2()

Prototype

```
SaAisErrorT saNtfStateChangeNotificationFilterAllocate_2(
     SaNtfHandleT ntfHandle,
     SaNtfStateChangeNotificationFilterT_2 *notificationFilter,
     SaUint16T numEventTypes,
     SaUint16T numNotificationObjects,
     SaUint16T numNotifyingObjects,
     SaUint16T numNotificationClassIds,
     SaUint16T numSourceIndicators,
     SaUint16T numChangedStates
);
```

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notificationFilter` - [out] A pointer to a structure of `SaNtfStateChangeNotificationFilterT_2` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfStateChangeNotificationFilterT_2` type is defined in Section 3.14.42 on page 71.

`numEventTypes` - [in] Number of event types in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationObjects` - [`in`] Number of notification objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotifyingObjects` - [`in`] Number of notifying objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationClassIds` - [`in`] Number of notification class ids in the notification filter. The `SaUint16T` type is defined in [2].

`numSourceIndicators` - [`in`] Number of source indicators in the notification filter. The `SaUint16T` type is defined in [2].

`numChangedStates` - [`in`] Number of changed states in the notification filter. The `SaUint16T` type is defined in [2].

Description

This function internally allocates memory for an attribute change notification filter and initializes the structure pointed to by the `notificationFilter` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notificationFilter` parameter. The pointers in the structure referred to by the `notificationFilter` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notificationFilter` parameter also contains the notification filter handle, which is used for subsequent calls to functions such as `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, 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 `ntfHandle` 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_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, `saNtfNotificationFilterFree()`

3.17.3 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 which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notificationFilter` - [out] A pointer to a structure of `SaNtfAlarmNotificationFilterT_2` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfAlarmNotificationFilterT` type is defined in Section 3.14.43 on page 72.

`numEventTypes` - [in] Number of event types in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationObjects` - [in] Number of notification objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotifyingObjects` - [in] Number of notifying objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationClassIds` - [in] Number of notification class ids in the notification filter. The `SaUint16T` type is defined in [2].

`numProbableCauses` - [`in`] Number of probable causes in the notification filter. The `SaUint16T` type is defined in [2].

`numPerceivedSeverities` - [`in`] Number of perceived severities in the notification filter. The `SaUint16T` type is defined in [2].

`numTrends` - [`in`] Number of trends in the notification filter. The `SaUint16T` type is defined in [2].

Description

This function internally allocates memory for an alarm notification filter and initializes the structure pointed to by the `notificationFilter` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notificationFilter` parameter. The pointers in the structure referred to by the `notificationFilter` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notificationFilter` parameter also contains the notification filter handle, which is used for subsequent calls to functions such as `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, 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 `ntfHandle` 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_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle ntfHandle was acquired before the cluster node left the cluster membership.

See Also

saNtfInitialize_3(), saNtfNotificationSubscribe_3(), saNtfNotificationReadInitialize_3(), saNtfNotificationReadNext_3(), saNtfNotificationFilterFree()

3.17.4 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 which was obtained by a previous invocation of saNtfInitialize_3() and which designates this particular initialization of the Notification Service. The SaNtfHandleT type is defined in Section 3.14.1.1 on page 48.

notificationFilter - [out] A pointer to a structure of SaNtfSecurityAlarmNotificationFilterT type. Memory for this structure

can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfSecurityAlarmNotificationFilterT` type is defined in Section 3.14.44 on page 73.

`numEventTypes` - [in] Number of event types in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationObjects` - [in] Number of notification objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotifyingObjects` - [in] Number of notifying objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationClassIds` - [in] Number of notification class ids in the notification filter. The `SaUint16T` type is defined in [2].

`numProbableCauses` - [in] Number of probable causes in the notification filter. The `SaUint16T` type is defined in [2].

`numSeverities` - [in] Number of severities in the notification filter. The `SaUint16T` type is defined in [2].

`numSecurityAlarmDetectors` - [in] Number of security alarm detectors in the notification filter. The `SaUint16T` type is defined in [2].

`numServiceUsers` - [in] Number of service users in the notification filter. The `SaUint16T` type is defined in [2].

`numServiceProviders` - [in] Number of service providers in the notification filter The `SaUint16T` type is defined in [2].

Description

This function internally allocates memory for a security alarm notification filter and initializes the structure pointed to by the `notificationFilter` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notificationFilter` parameter. The pointers in the structure referred to by the `notificationFilter` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notificationFilter` parameter also contains the notification filter handle, which is used for subsequent calls to functions such as `saNtfNotificationSubscribe_3()`,

`saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, 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 `ntfHandle` 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_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, `saNtfNotificationFilterFree()`

3.17.4.1 saNtfMiscellaneousNotificationFilterAllocate()

Prototype

```
SaAisErrorT saNtfMiscellaneousNotificationFilterAllocate(
      SaNtfHandleT ntfHandle,
      SaNtfMiscellaneousNotificationFilterT *notificationFilter,
      SaUint16T numEventTypes,
      SaUint16T numNotificationObjects,
      SaUint16T numNotifyingObjects,
      SaUint16T numNotificationClassIds,
);
```

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`notificationFilter` - [out] A pointer to a structure of `SaNtfMiscellaneousNotificationFilterT` type. Memory for this structure can be on the stack or the heap, that is, it has to be allocated by the invoking process. The `SaNtfMiscellaneousNotificationFilterT` type is defined in Section 3.14.45 on page 74.

`numEventTypes` - [in] Number of event types in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationObjects` - [in] Number of notification objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotifyingObjects` - [in] Number of notifying objects in the notification filter. The `SaUint16T` type is defined in [2].

`numNotificationClassIds` - [in] Number of notification class ids in the notification filter. The `SaUint16T` type is defined in [2].

Description

This function internally allocates memory for a miscellaneous notification filter and initializes the structure pointed to by the `notificationFilter` parameter. The values of the function parameters indicating a size or the number of array elements are copied to the corresponding attributes in the structure pointed to by the `notificationFilter` parameter. The pointers in the structure referred to by the `notificationFilter` parameter are initialized to point to fields in the internal data structure. When this function completes successfully, the structure pointed to by the `notificationFilter` parameter also contains the notification filter handle, which is used for subsequent calls to functions such as `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, 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 `ntfHandle` 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_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfInitialize_3()`, `saNtfNotificationSubscribe_3()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`, `saNtfNotificationFilterFree()`

3.17.4.2 saNtfNotificationFilterFree()

Prototype

```
SaAisErrorT saNtfNotificationFilterFree(
      SaNtfNotificationFilterHandleT notificationFilterHandle
);
```

Parameters

`notificationFilterHandle` - [in] notification filter handle. The `SaNtfNotificationFilterHandleT` type is defined in Section 3.14.1.3 on page 48.

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.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `notificationFilterHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfObjectCreateDeleteNotificationFilterAllocate()`,
`saNtfAttributeChangeNotificationFilterAllocate()`,
`saNtfStateChangeNotificationFilterAllocate_2()`,
`saNtfAlarmNotificationFilterAllocate()`,
`saNtfSecurityAlarmNotificationFilterAllocate()`,
`saNtfMiscellaneousNotificationFilterAllocate()`

3.17.5 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. Invoking `saNtfInitialize_3()` to initialize the Notification Service for the calling process and to supply callback functions for handling received notifications.
2. Allocating memory for the notification filter contents by invoking one or several of the allocation functions described in Section 3.17.1 on page 117.
3. Filling in the notification filter fields of the structure or structures allocated in the previous step.
4. Calling `saNtfNotificationSubscribe_3()` with the filter handles returned in step 2.
5. Releasing the allocated memory for the notification filter contents by calling the `saNtfNotificationFilterFree()` function (if not needed otherwise).
6. Eventually deleting the subscription by invoking `saNtfNotificationUnsubscribe_2()` when no more notifications are to be processed.

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 lower, but must not be higher 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.

3.17.5.1 saNtfNotificationSubscribe_3()

Prototype

```
SaAisErrorT saNtfNotificationSubscribe_3(
      const SaNtfNotificationTypeFilterHandlesT_3
           *notificationFilterHandles,
      SaNtfSubscriptionIdT subscriptionId
);
```

Parameters

`notificationFilterHandles` - [in] A pointer to the handles of the notification-type-specific filters previously returned by the allocation functions. A filter must have

been allocated for at least one notification type. If more than one handle is used in the structure, all handles must have been generated for the same instance of the Notification Service (that is, with the same `SaNtfHandleT` value). Notification types for which no subscription is to be established must have set their corresponding field in `notificationFilterHandles` to `SA_NTF_FILTER_HANDLE_NULL`. The `SaNtfNotificationTypeFilterHandlesT_3` type is defined in Section 3.14.49 on page 75.

`subscriptionId` - [`in`] Used to identify a particular subscription within the context of the subscriber application. It must be unique for all subscriptions established for the instance of the Notification Service that 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 established several subscriptions. The subscription id has to be used when unsubscribing with `saNtfNotificationUnsubscribe_2()`. The `SaNtfSubscriptionIdT` type is defined in Section 3.14.38 on page 69.

Description

The `saNtfNotificationSubscribe_3()` function enables a process to subscribe for notifications by registering one or more filters referred to by `notificationFilterHandles`.

Notifications are delivered by the invocation of the `SaNtfNotificationCallbackT_3` callback function, which must have been supplied when the process called `saNtfInitialize_3()`.

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 a call to the `saNtfNotificationSubscribe_3()` function, the process may safely free the filters by invoking `saNtfNotificationFilterFree()` or use them for other calls to the Consumer API functions (that is, for `saNtfNotificationReadInitialize_3()` or `saNtfNotificationSubscribe_3()`).

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 - Not all handles pointed to by notificationFilterHandles refer to the same instance of the Notification Service or one or more handles are invalid. A handle is invalid if it is corrupted, or it is uninitialized, or it 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_EXIST - A subscription with the value of subscriptionId already exists for this instance of the Notification Service (that is, the SaNtfHandleT value that was used to allocate the filters).

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but one or more of the handles referred to by the notificationFilterHandles pointer were acquired before the cluster node left the cluster membership.

See Also

saNtfInitialize_3(), saNtfDispatch(),
saNtfNotificationUnsubscribe_2(),
saNtfObjectCreateDeleteNotificationFilterAllocate(),
saNtfAttributeChangeNotificationFilterAllocate(),
saNtfStateChangeNotificationFilterAllocate_2(),
saNtfAlarmNotificationFilterAllocate(),
saNtfSecurityAlarmNotificationFilterAllocate(),
saNtfMiscellaneousNotificationFilterAllocate()

3.17.5.2 saNtfNotificationUnsubscribe_2()

Prototype

```
SaAisErrorT saNtfNotificationUnsubscribe_2(
      SaNtfHandleT ntfHandle,
      SaNtfSubscriptionIdT subscriptionId
);
```

Parameters

`ntfHandle` - [in] The handle which was obtained by a previous invocation of `saNtfInitialize_3()` and which designates this particular initialization of the Notification Service. The `SaNtfHandleT` type is defined in Section 3.14.1.1 on page 48.

`subscriptionId` - [in] Subscription identifier that was previously passed to `saNtfNotificationSubscribe_3()`. The `SaNtfSubscriptionIdT` type is defined in Section 3.14.38 on page 69.

Description

The `saNtfNotificationUnsubscribe_2()` function deletes the subscription previously established in an invocation of the `saNtfNotificationSubscribe_3()` function.

Queued notifications which match the filter settings passed at subscription time and which have not been delivered to the subscriber by one of the notification callbacks are implicitly discarded. If the subscriber wants to avoid this behavior, the subscriber must establish a new subscription before the old one is deleted.

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 freed.

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 the instance of the Notification Service which is identified by the `SaNtfHandleT` value, and which was used to allocate the filters for the subscription.

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `ntfHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfNotificationSubscribe_3()`

3.17.5.3 SaNtfNotificationCallbackT_3

Prototype

```
typedef void (*SaNtfNotificationCallbackT_3)(
     SaNtfSubscriptionIdT subscriptionId,
     const SaNtfNotificationsT_3 *notification
);
```

Parameters

`subscriptionId` - [in] An identifier that the subscriber supplied in an `saNtfNotificationSubscribe_3()` invocation and that enables the subscriber to determine which subscription resulted in the delivery of the notification. The `SaNtfSubscriptionIdT` type is defined in Section 3.14.38 on page 69.

`notification` - [in] A pointer to the notification delivered by this callback. The `SaNtfNotificationsT_3` type is defined in Section 3.14.50 on page 76.

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 calling `saNtfDispatch()` with the handle `ntfHandle` that was specified when the notification filters (used for the subscription) were allocated by invoking the `saNtf<notification type>FilterAllocate()` functions.

This callback is invoked for each subscription matching a particular notification. As an example, if a process has two subscriptions for notifications, and a particular notification matches the filters of both subscriptions, the callback is invoked twice, that is, once for each subscription.

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

Return Values

None

See Also

`saNtfNotificationSubscribe_3()`, `saNtfNotificationFree()`, `saNtfDispatch()`

3.17.5.4 SaNtfNotificationDiscardedCallbackT

Prototype

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

Parameters

`subscriptionId` - [in] An identifier that a process supplied in an `saNtfNotificationSubscribe_3()` invocation and that enables it to determine for which subscription notifications have been discarded. The `SaNtfSubscriptionIdT` type is defined in Section 3.14.38 on page 69.

`notificationType` - [in] The notification type of the discarded notifications. The `SaNtfNotificationTypeT` type is defined in Section 3.14.3 on page 49.

`numberDiscarded` - [in] The number of discarded notifications. The `SaUint32T` type is defined in [2].

`discardedNotificationIdentifiers` - [in] A pointer to a 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 list contains as many elements as indicated by `numberDiscarded`. The `SaNtfIdentifierT` type is defined in Section 3.14.11 on page 53.

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 calling `saNtfDispatch()` with the handle `ntfHandle` that was specified when the notification filters (used for the subscription) were allocated by invoking the `saNtf<notification type>FilterAllocate()` functions. Unless `discardedNotificationIdentifiers` is NULL, the subscriber can use the Reader API to retrieve the notifications.

If the subscriber wants to keep the contents of the list pointed to by `discardedNotificationIdentifiers` for processing after it has returned from this callback, it has to copy the list before returning. The pointer `discardedNotificationIdentifiers` and the contents of the list to which it points are fully controlled by the Notification Service library. In particular, the subscriber must not change the contents of or call `free()` for the list pointed to by `discardedNotificationIdentifiers`.

Refer also to Section 3.6.1.

Return Values

None

See Also

`saNtfNotificationSubscribe_3()`, `saNtfDispatch()`, `saNtfNotificationReadInitialize_3()`, `saNtfNotificationReadNext_3()`

3.17.6 Operations of the Reader API

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

1. Allocating memory for the notification filter contents by invoking one or several of the allocation functions that are described in Section 3.17.1 on page 117.
2. Filling in the notification filter fields of the structure or structures allocated in the previous step.
3. Calling `saNtfNotificationReadInitialize_3()` with the filter handles returned from step 1. to obtain a read handle.
4. Releasing the memory allocated for the filters by invoking the `saNtfNotificationFilterFree()` function (if not needed otherwise).
5. Calling `saNtfNotificationReadNext_3()` with the read handle returned from step 3. and specifying the search direction.
6. Releasing the memory allocated for the notification by invoking the `saNtfNotificationFree()` function (if not needed any longer).
7. Releasing any resources bound to the read handle by invoking the `saNtfNotificationReadFinalize()` function.

Step 5. and 6. may be repeated multiple times.

3.17.6.1 saNtfNotificationReadInitialize_3()

Prototype

```
SaAisErrorT saNtfNotificationReadInitialize_3(
      const SaNtfSearchCriteriaT *searchCriteria,
      const SaNtfNotificationTypeFilterHandlesT_3
            *notificationFilterHandles,
      SaNtfReadHandleT *readHandle
);
```

Parameters

`searchCriteria` - [in] In addition to the filter criteria, the structure pointed to by the `searchCriteria` parameter can 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`, the search will start with the oldest existing notification. The `SaNtfSearchCriteriaT` type is defined in Section 3.14.47 on page 74.

`notificationFilterHandles` - [in] A pointer to a structure containing handles of the notification-type-specific filters previously generated by the filter allocate functions. A filter must have been allocated for at least one notification type. If two or more handles are used in the structure, all handles must have been generated for the same instance of the Notification Service (that is, with the same `SaNtfHandleT` value). For each notification type in which the caller is not interested, the corresponding field in the structure referred to by `notificationFilterHandles` must be set to `SA_NTF_FILTER_HANDLE_NULL`. The `SaNtfNotificationTypeFilterHandlesT_3` type is defined in Section 3.14.49 on page 75.

`readHandle` - [out] A pointer to the read handle returned by the function. The read handle is to be used for subsequent calls to `saNtfNotificationReadNext_3()`. When no more notifications are to be read for the given filter criteria, the application must free its related resources by invoking `saNtfNotificationReadFinalize()`. The `SaNtfReadHandleT` type is defined in Section 3.14.1.4 on page 48.

Description

This function is used to initialize reading logged notifications according to the search criteria specified in the structure pointed to by `searchCriteria` and filters referred to by `notificationFilterHandles`.

The search criteria can be used to specify a point in time or a notification identifier where reading of notifications should start when `saNtfNotificationReadNext_3()` is called the first time with the new read handle. If the `searchMode` field in the structure referred to by `searchCriteria` is set to `SA_NTF_SEARCH_ONLY_FILTER`, only the filters referred to by the pointer `notificationFilterHandles` are used.

The typical way to read a chronologically ordered list of notifications is to specify a starting point and filter criteria with `saNtfNotificationReadInitialize_3()` and to read notifications with a sequence of calls to `saNtfNotificationReadNext_3()`.

This function is used to initialize reading logged notifications according to the search criteria specified in the structure pointed to by `searchCriteria` and filters referred to by `notificationFilterHandles`. The logged notifications are ordered chronologically, based on the time when a notification is written to the log file.

The search criteria is used to specify a point in time or a notification identifier from which reading of notifications should start when `saNtfNotificationReadNext_3()` is called the first time with the new read handle.

If the searchMode field in the structure referred to by searchCriteria is set to SA_NTF_SEARCH_ONLY_FILTER, only the filters referred to by the pointer notificationFilterHandles are used.
The filters referred to by notificationFilterHandles may specify additional filter criteria. For each notification type in which the caller is interested, filter criteria have to be specified (that is, a filter handle has to be set in the structure referred to by 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.

It is the responsibility of the process to free the notification filters by invoking the saNtfNotificationFilterFree() function if they are no longer needed. After a call to the saNtfNotificationReadInitialize_3()function, the process may safely free the filters by invoking saNtfNotificationFilterFree() or use them for other calls to the Consumer API functions (that is, saNtfNotificationReadInitialize_3() or saNtfNotificationSubscribe_3()).

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 - Not all handles pointed to by notificationFilterHandles refer to the same instance of the Notification Service or one or more handles are invalid. A handle is invalid if it is corrupted, or it is uninitialized, or it 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_NOT_SUPPORTED - In the structure pointed to by notificationFilterHandles, at least one handle has been set for a notification type that is not supported by this implementation.

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

SA_AIS_ERR_VERSION - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

SA_AIS_ERR_UNAVAILABLE - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but one or more of the handles referred to by the `notificationFilterHandles` pointer were acquired before the cluster node left the cluster membership.

See Also

`saNtfNotificationReadNext_3()`, `saNtfNotificationReadFinalize()`, `saNtfObjectCreateDeleteNotificationFilterAllocate()`, `saNtfAttributeChangeNotificationFilterAllocate()`, `saNtfStateChangeNotificationFilterAllocate_2()`, `saNtfAlarmNotificationFilterAllocate()`, `saNtfSecurityAlarmNotificationFilterAllocate()`, `saNtfMiscellaneousNotificationFilterAllocate()`

3.17.6.2 saNtfNotificationReadNext_3()

Prototype

```
SaAisErrorT saNtfNotificationReadNext_3(
     SaNtfReadHandleT readHandle,
     SaNtfSearchDirectionT searchDirection,
     SaNtfNotificationsT_3 *notification
);
```

Parameters

`readHandle` - [in] The read handle which was previously obtained by a call to `saNtfNotificationReadInitialize_3()`. The `SaNtfReadHandleT` type is defined in Section 3.14.1.4 on page 48.

`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_3()`, this parameter is ignored. The `SaNtfSearchDirectionT` type is defined in Section 3.14.48 on page 75.

`notification` - [out] A pointer to the notification returned by the function. The `notificationType` field determines which of the fields in the union is valid, that is, which field actually contains the notification. This variable can be on the stack or heap, that is, 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()`. The `SaNtfNotificationsT_3` type is defined in Section 3.14.50 on page 76.

Description

This function is used to read chronologically ordered logged notifications. Reading must have been initialized with `saNtfNotificationReadInitialize_3()`. As many as wanted notifications may then be read with a sequence of calls to `saNtfNotificationReadNext_3()`.

When this function is called for the first time after the read handle has been obtained from `saNtfNotificationReadInitialize_3()`, it will ignore the `searchDirection` parameter and will use only the search and filter criteria passed to `saNtfNotificationReadInitialize_3()`. For subsequent invocations with

the same read handle, this function uses `searchDirection` and the filter criteria previously passed to `saNtfNotificationReadInitialize_3()`.

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_3()`, 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` - The 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` - No notification matches the given criteria.

`SA_AIS_ERR_VERSION` - The invoked function is not supported in the version specified in the call to initialize this instance of the Notification Service library.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `readHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfNotificationReadInitialize_3()`,
`saNtfNotificationReadFinalize()`,
`saNtfObjectCreateDeleteNotificationFilterAllocate()`,

```
saNtfAttributeChangeNotificationFilterAllocate(),
saNtfStateChangeNotificationFilterAllocate_2(),
saNtfAlarmNotificationFilterAllocate(),
saNtfSecurityAlarmNotificationFilterAllocate(),
saNtfMiscellaneousNotificationFilterAllocate()
```

3.17.6.3 saNtfNotificationReadFinalize()

Prototype

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

Parameters

`readHandle` - [in] The read handle which was obtained by a previous invocation of `saNtfNotificationReadInitialize_3()`. The `SaNtfReadHandleT` type is defined in Section 3.14.1.4 on page 48.

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_3()`.

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 `readHandle` is invalid, since it is corrupted, uninitialized, or has already been freed.

`SA_AIS_ERR_UNAVAILABLE` - The operation requested in this call is unavailable on this cluster node due to one of the two reasons:

- the cluster node has left the cluster membership;
- the cluster node has rejoined the cluster membership, but the handle `readHandle` was acquired before the cluster node left the cluster membership.

See Also

`saNtfNotificationReadNext_3()`,
`saNtfNotificationReadInitialize_3()`,
`saNtfObjectCreateDeleteNotificationFilterAllocate()`,
`saNtfAttributeChangeNotificationFilterAllocate()`,
`saNtfStateChangeNotificationFilterAllocate_2()`,
`saNtfAlarmNotificationFilterAllocate()`,
`saNtfSecurityAlarmNotificationFilterAllocate()`,
`saNtfMiscellaneousNotificationFilterAllocate()`

4 Notification Service UML Information Model

The Notification Service Information Model is described in UML and has been organized in a UML class diagram.

The Notification Service UML model is implemented by the SA Forum IMM Service ([3]). For further details on this implementation, refer to the SA Forum Overview document ([1]).

The Notification Service UML class diagram has five classes, which show the contained attributes and the administrative operations (if any) applicable on these classes.

4.1 DN Format for Notification Service UML Classes

Table 15 DN Formats for Objects of Notification Service Classes

Object Class	DN Formats for Objects of the Class
`SaNtfStaticFilter`	"`safStaticFilter=...`, `safApp=safNtfService`"
`SaNtfObjectCreate DeleteFilterElementSet`	"`safOCDFilterElementSet=...`, `safStaticFilter=...`, `safApp=safNtfService`"
`SaNtfAttributeChange FilterElementSet`	"`safACFilterElementSet=...`, `safStaticFilter=...`, `safApp=safNtfService`"
`SaNtfStateChangeFilter ElementSet`	"`safSCFilterElementSet=...`, `safStaticFilter=...`, `safApp=safNtfService`"
`SaNtfMiscellaneousFilter ElementSet`	"`safMiscFilterElementSet=...`, `safStaticFilter=...`, `safApp=safNtfService`"

4.2 Notification Service UML Classes

The five classes of the Notification Service UML model are:

- `SaNtfStaticFilter` —This configuration object class defines filters for static suppression of notifications.
- `SaNtfObjectCreateDeleteFilterElementSet` —This configuration object class defines filter element sets corresponding to the attributes of object create/delete notifications.

- `SaNtfAttributeChangeFilterElementSet` — This configuration object class defines filter element sets corresponding to the attributes of attribute change notifications.
- `SaNtfStateChangeFilterElementSet` — This configuration object class defines filter element sets corresponding to the attributes of state change notifications.
- `SaNtfMiscellaneousFilterElementSet` — This configuration object class defines filter element sets corresponding to the attributes of a miscellaneous notification.

FIGURE 2 shows these classes. A description of each attribute of these classes may be found in the XMI file (see [5]). For additional details, refer to the SA Forum Overview document ([1]).

Note: The following restriction exists for the configuration of static notification suppression filters:
An `SaNtfStaticFilter` object cannot be successfully deleted when it is active. It has to be deactivated before it can be deleted.

To configure the `saNtf<type>NotificationClassIds` attributes, the `SaStringT` type (defined in [2]) is used. The contents of such an attribute are the decimal values for the `vendorId`, `majorId`, and `minorId` fields of the `SaNtfClassIdT` structure in this order and separated by a '.'. For example, the notification class id for the NTF Consumer Slow notification is: "18568.10.103".

FIGURE 2 UML Class Diagram

5 Notification Service Administration API

5.1 Notification Service Administration API Model

5.1.1 Notification Service Administration API Basics

This chapter describes the administrative API that the IMM Service (see [3]) exposes on behalf of the Notification Service to a system administrator. This API is described using a 'C' API syntax.

The main clients of this administrative API are system management applications and SNMP agents (see [14]), which typically convert system administration commands (invoked from a management station) to the correct administrative API sequence to yield the desired result that is expected upon execution of the system administration command.

The Notification Service administrative API is applicable to the entities that are controlled by the Notification Service such as the suppression filters.

This API will be exposed by the IMM Service Object Management library.

5.2 Include File and Library Name

The appropriate IMM Service header file and the Notification Service header file must be included in the source of an application using the Notification Service administration API. For the name of the IMM Service header file, see [3].

5.3 Type Definitions

The specification of Notification Service administration API requires the following types, in addition to the ones already described.

5.3.1 SaNtfAdminOperationIdT

```
typedef enum {
     SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER        = 1,
     SA_NTF_ADMIN_DEACTIVATE_STATIC_SUPPRESSION_FILTER      = 2
} SaNtfAdminOperationIdT;
```

5.4 Notification Service Administration API

As explained above, the administrative API shall be exposed by the IMM (see [3]) Service library. The IMM Service API `saImmOmAdminOperationInvoke_3()` or `saImmOmAdminOperationInvokeAsync_3()` shall be invoked with the appropriate `operationId` (see Section 5.3.1) and `objectName` to execute a particular administrative operation. In the following section, the administrative API is described with the assumption that the SA Forum Notification Service is an object implementer (runtime owner) for the various administrative operations that will be initiated as a consequence of invoking the `saImmOmAdminOperationInvoke_3()` or the `saImmOmAdminOperationInvokeAsync_3()` functions with the appropriate `operationId` (see Section 5.3.1) on the filter object for static notification suppression designated by the name to which `objectName` points.

The API syntax of the administrative API shall use only the corresponding enumeration value for the `operationId` (see Section 5.3.1) for administrative operations on the filter object for static notification suppression.

The return values explained in the section below shall be passed in the `operationReturnValue` parameter, which is provided by the invoker of the `saImmOmAdminOperationInvoke_3()` or the `saImmOmAdminOperationInvokeAsync_3()` function to obtain return codes from the object implementer (Notification Service in this case).

5.4.1 SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER

Parameters

`operationId` - [in] = `SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER`

`objectName` - [in] Pointer to the LDAP name of the `SaNtfStaticFilter` object to be activated. The initial RDN type must be `safStaticFilter`. For SA Forum naming conventions and rules, see [2].

Description

This administrative operation activates the specified filter for static notification suppression. The filter becomes effective, and any notifications matching the filter are suppressed.

Return Values

SA_AIS_OK - The function completed successfully.

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 client may retry later. This error generally should be returned in cases for which the requested action is valid but not currently possible, probably because another administrative operation is acting upon the logical entity on which the administrative operation is invoked.

SA_AIS_ERR_NO_MEMORY - The Notification Service or a library 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 - The logical entity identified by the name to which objectName points does not exist in the configuration repository.

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported by the type of entity denoted by the name to which objectName points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of the filter, as the filter is already active.

See Also

SA_NTF_ADMIN_DEACTIVATE_STATIC_SUPPRESSION_FILTER

5.4.2 SA_NTF_ADMIN_DEACTIVATE_STATIC_SUPPRESSION_FILTER

Parameters

operationId - [in] = SA_NTF_ADMIN_DEACTIVATE_STATIC_SUPPRESSION_FILTER

objectName - [in] Pointer to the LDAP name of the SaNtfStaticFilter object to be deactivated. The initial RDN type must be safStaticFilter. For SA Forum naming conventions and rules, see [2].

Description

This administrative operation deactivates the specified filter for static notification suppression. If the filter is already active when this operation is executed (due to a previously executed SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER

operation), it becomes ineffective, and notifications are no longer suppressed by the filter.

Return Values

SA_AIS_OK - The function completed successfully.

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 client may retry later. This error generally should be returned in cases for which the requested action is valid but not currently possible, probably because another administrative operation is acting upon the logical entity on which the administrative operation is invoked.

SA_AIS_ERR_NO_MEMORY - The Notification Service or a library 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 - The logical entity identified by the name to which objectName points does not exist in the configuration repository.

SA_AIS_ERR_NOT_SUPPORTED - This administrative operation is not supported by the type of entity denoted by the name to which objectName points.

SA_AIS_ERR_NO_OP - The invocation of this administrative operation has no effect on the current state of the filter, as the filter is currently not active.

See Also

SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER

6 Alarms and Notifications

The Notification Service produces alarms and notifications to convey important information regarding the operational and functional state of the objects under its control to an administrator or a management system.

These reports vary in perceived severity and include alarms, which potentially require an operator intervention, and notifications which signify important state or object changes. A management entity should regard notifications, but they do not necessarily require an operator intervention.

The vehicle to be used for producing alarms and notifications is the Notification Service itself. The various notifications are partitioned into categories as described in the Notification Service.

In some cases, this specification uses the word “Unspecified” for values of attributes that the vendor is at liberty to set to whatever makes sense in the vendor’s context, and the SA Forum has no specific recommendation regarding such values. Such values are generally optional from the CCITT Recommendation X.733 perspective (see [12]).

6.1 Setting Common Attributes

The following attributes of the notifications presented in Section 6.2 are not shown in their description, as the generic description presented here applies to all of them:

- Notification Id - This attribute is either implicitly set by the Notification Service when a notification is sent by invoking the `saNtfNotificationSend()` function or is provided by the caller when the notification is sent by invoking the `saNtfNotificationSendWithId()` function.
- Notifying Object - DN of the entity generating the notification. This name must conform to the SA Forum AIS naming convention and must contain at least the `safApp` RDN value portion of the DN set to the specified standard RDN value of the SA Forum AIS Service generating the notification, which in this case is “`safApp=safNtfService`". For details on the SA Forum AIS naming convention, refer to [2].
- Event Time - This attribute contains the time when the Notification Service detected the event leading to the notification.
- Correlated Notifications - Correlation ids are supplied to correlate notifications that have been generated because of a related cause. The correlated notifications attribute should include

- in the first position the root notification identifier of the related tree of notifications as described in this specification;
- in the second position the parent notification identifier of the same tree;
- in the third position the notification identifier of the sibling notification, if any. This sibling notification is the opening pair of the current notification such as the alarm that is being cleared or the start of an administrative operation or a configuration change that has ended.

If any of these notifications is unknown, the SA_NTF_IDENTIFIER_UNUSED value must be used. This value may be omitted in trailing positions.

The following note applies to all notifications presented in Section 6.2:

- Notification Class Identifier - The vendorId field of the SaNtfClassIdT data structure must be set to SA_NTF_VENDOR_ID_SAF, and the majorId field must be set to SA_SVC_NTF (as defined in the SaServicesT enumeration in [2]) for all notifications that follow the standard formats described in this specification. The minorId field will vary based on the specific notification.

6.2 Notification Service Notifications

The following subsections describe the notifications that a Notification Service implementation shall produce.

6.2.1 Notification Service Alarms

The Notification Service does not issue any alarms at the time of publication of this specification.

6.2.2 Notification Service State Change Notifications

6.2.2.1 Static Suppression Filter Activated

Description

A filter for static suppression of notifications has been activated by the execution of the SA_NTF_ADMIN_ACTIVATE_STATIC_SUPPRESSION_FILTER administrative operation.

Table 16 Static Suppression Filter Activated Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	The LDAP DN of the `SaNtfStaticFilter` object that has been activated
Notification Class Identifier	NTF-Internal	`minorId` = `SA_NTF_NTFID_STATIC_FILTER_ACTIVATED`, see Section 3.14.55 on page 77
Additional Text	Optional	"Filter for static suppression <`filter name`> activated."
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	`SA_NTF_MANAGEMENT_OPERATION`
Changed State Attribute ID	Optional	`SA_NTF_STATIC_FILTER_STATE`
Old Attribute Value	Optional	Unspecified
New Attribute Value	Mandatory	`SA_NTF_STATIC_FILTER_STATE_ACTIVE`

6.2.2.2 Static Suppression Filter Deactivated

Description

A filter for static suppression of notifications has been deactivated by the execution of the SA_NTF_ADMIN_DEACTIVATE_STATIC_SUPPRESSION_FILTER administrative operation.

Table 17 Static Suppression Filter Deactivated Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	SA_NTF_OBJECT_STATE_CHANGE
Notification Object	Mandatory	The LDAP DN of the SaNtfStaticFilter object (as specified in Section 4.1) that has been activated
Notification Class Identifier	NTF-Internal	minorId = SA_NTF_NTFID_STATIC_FILTER_DEACTIVATED, see Section 3.14.55 on page 77
Additional Text	Optional	"Filter for static suppression <filter name> deactivated."
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	SA_NTF_MANAGEMENT_OPERATION
Changed State Attribute ID	Optional	SA_NTF_STATIC_FILTER_STATE
Old Attribute Value	Optional	SA_NTF_STATIC_FILTER_STATE_ACTIVE
New Attribute Value	Mandatory	SA_NTF_STATIC_FILTER_STATE_INACTIVE

6.2.2.3 Subscriber Consuming Too Slowly

Description

A notification subscriber does not consume or does not consume fast enough the notifications that have been forwarded to it. No more notifications can be forwarded to the subscriber. The Notification Service informs the subscriber about discarded notifications by invoking the `SaNtfNotificationDiscardedCallbackT` function.

Table 18 Subscriber Consuming Too Slowly Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the component that does not consume or does not consume fast enough notifications. If the notification object in this notification (the subscriber of the NTF Service) is a part of a component under the control of the Availability Management Framework, this field should contain the name of that component, which is a LDAP DN (in this case, it is expected that in future, it would be mandatory to pass the LDAP DN of the component). If the notification object is not under the control of the Availability Management Framework, it is highly recommended that the name of the notification object follows the SA Forum naming convention (as defined in [2]) and, in particular, follow the LDAP DN structure of a component.
Notification Class Identifier	NTF-Internal	`minorId` = `SA_NTF_NTFID_CONSUMER_SLOW`, see Section 3.14.55 on page 77
Additional Text	Optional	“Subscriber <LDAP DN of the component> does not consume or does not consume fast enough notifications and cannot accept any more notifications.” The <LDAP DN of the component> has the same value as the notification object attribute.
Additional Information	Optional	Unspecified

Table 18 Subscriber Consuming Too Slowly Notification (Continued)

NTF Attribute Name	Mandatory/ Optional	Specified Value
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_NTF_SUBSCRIBER_STATE`
Old Attribute Value	Optional	Unspecified
New Attribute Value	Mandatory	`SA_NTF_SUBSCRIBER_STATE_FORWARD_NOT_OK`

6.2.2.4 Subscriber Consumes Fast Enough

Description

A notification subscriber previously did not consume or did not consume fast enough notifications that were forwarded to it. Now, the Notification Service can again forward notifications to this subscriber.

Table 19 Subscriber Consumes Fast Enough Notification

NTF Attribute Name	Mandatory/ Optional	Specified Value
Event Type	Mandatory	`SA_NTF_OBJECT_STATE_CHANGE`
Notification Object	Mandatory	LDAP DN of the component that did not consume or did not consume fast enough notifications. If the notification object in this notification (the subscriber of the NTF Service) is a part of a component under the control of the Availability Management Framework, this field should contain the name of that component, which is a LDAP DN (in this case, it is expected that in future, it would be mandatory to pass the LDAP DN of the component). If the notification object is not under the control of the Availability Management Framework, it is highly recommended that the name of the notification object follows the SA Forum naming convention (as defined in [2]) and, in particular, follow the LDAP DN structure of a component.
Notification Class Identifier	NTF-Internal	`minorId` = `SA_NTF_NTFID_CONSUMER_FAST_ENOUGH`, see Section 3.14.55 on page 77
Additional Text	Optional	“Notifications can again be forwarded to subscriber <LDAP DN of the component>.” The <LDAP DN of the component> has the same value as the notification object attribute.
Additional Information	Optional	Unspecified
Source Indicator	Mandatory	`SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION`
Changed State Attribute ID	Optional	`SA_NTF_SUBSCRIBER_STATE`
Old Attribute Value	Optional	`SA_NTF_SUBSCRIBER_STATE_FORWARD_NOT_OK`
New Attribute Value	Mandatory	`SA_NTF_SUBSCRIBER_STATE_FORWARD_OK`

Appendix A Trap OID Mapping

This appendix presents a possible implementation for a mapping table that is no normative part or the Notification Service specification. The description is high level.

For the SNMP (see [14]) interface, it is important that notification class identifiers (NCIs) can be mapped to OIDs of SNMP traps. This will allow an 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 performs a table lookup by using the notification class identifier as an index into the mapping table.

Table 20 Trap OID 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 that is used by default when no record is found for a particular notification class identifier.
Trap OID	string	OID of the SNMP trap in dot-notation.

Appendix B Internationalization

This appendix presents a possible implementation for a mapping table that is no normative part or the Notification Service specification. The description is high level.

Optionally, internationalized textual information for each notification class identifier (NCI) 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 notification class identifier and one catalog type, only an entry may exist.

For POSIX message catalogs, the following configuration data fields are relevant:

- Catalog
- Format
- Set ID
- Message ID

For GNU message catalogs, the following configuration data fields are relevant:

- Catalog
- Format

Set ID and Message ID are not relevant for the GNU message catalogs, 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.

Table 21 Internationalization 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 internationalization data that is used by default when no record is found for a particular notification class identifier.
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 that 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, the SA Forum does not define any internationalization data but provides only the underlying mechanisms. All message catalogs and related configuration entries have to be provided by implementations.

The format string may contain keywords determining the 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 refer to the notification object. For list-type parameters, like specific problems or object attributes, each list element is referred to by C style array syntax, for instance, “object ${notificationObject} created with `xyz` attribute value ${objectAttributes[0].attributeValue}”, which refers to 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 allowed to include elements of data type SA_NTF_VALUE_ARRAY or SA_NTF_VALUE_BINARY in the format text.

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

Table 22 Mapping Keywords to Notification Parameters

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
thresholdInformation.thresholdId thresholdInformation.thresholdValue thresholdInformation.thresholdHysteresis thresholdInformation.observedValue thresholdInformation.armTime	Threshold Information

Table 22 Mapping Keywords to Notification Parameters (Continued)

Keywords	Notification Parameter
`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 C API Usage Examples

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

C.1 Producer Side (Example Function) – Object Create/Delete Notification

```
/* Send a notification about the creation of an object of type AbcObject that 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 by means of 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}".
*/

static SaNtfEventTypeBitmapT mySuppressedEventTypeBitmap;
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 NTF 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 by the attributeId field;
   the list of possible values of attributeId is NCI-specific 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 by the attributeId field;
   the list of possible values of attributeId is NCI-specific 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;
}

/* This callback stores the bitmap indicating which of the notification event types
   are currently completely statically suppressed. The bitmap is used in the
   sendAbcCreateNotification() function to determine whether an object create
   notification makes sense to be sent at all */

void myStaticSuppressionFilterSetCallback(
        SaNtfHandleT ntfHandle,
        SaNtfEventTypeBitmapT eventTypeBitmap)
{
        mySuppressedEventTypeBitmap = eventTypeBitmap;
}
```

An example for calling the above `sendAbcCreateNotification()` function:

```
        SaAisErrorT ret;
        SaVersionT version;
        SaNtfHandleT ntfHandle;
        SaNtfCallbacksT_3 ntfCallbacks;
        SaNtfIdentifierT ntfId;
        SaNameT name;
        SaNtfHandleT myNtfInstHandle;

        /* … */

        /* set up callback informing about static suppression of all notifications
         of particular notification types */
        ntfCallbacks.saNtfStaticSuppressionFilterSetCallback =
        myStaticSuppressionFilterSetCallback;

        /* then initialize the library instance - this will implicitly call
         myStaticSuppressionFilterSetCallback */
        ret = saNtfInitialize_3(&ntfHandle, &ntfCallbacks, &version);
```

```
if (ret != SA_AIS_OK) {
       /* could not initialize the Notification Service library */
       exit (1);
}

/* … */

/* this possibly calls myStaticSuppressionFilterSetCallback */
ret = saNtfDispatch(ntfHandle, SA_DISPATCH_ALL);
if (ret != SA_AIS_OK) {
       /* in case of error, mySuppressedEventTypeBitmap might not be up-to-date
        - can reset it manually for SA_AIS_ERR_TRY_AGAIN */
       if (ret == SA_AIS_ERR_TRY_AGAIN)
              mySuppressedEventTypeBitmap = 0;
       else
              exit(2);
}

/* check whether object create event types are currently completely
 suppressed - if not, send out the notification */
if ( !(mySuppressedEventTypeBitmap & SA_NTF_OBJECT_CREATION_BIT) ) {
       /* 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);
}
```

C.2 Producer Side (Example Function) – Attribute Change Notification

```
/* Send a notification about the modification of an object of type AbcObject that 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
by means of 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 NTF 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);
```

C.3 Producer Side (Example Function) – State Change Notification

```
/* Send a notification about the state changes of an object of type AbcObject that has two
state attributes: operational state and usage state. In this example, the state change
notification is correlated to a single previous notification by means of 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_3 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_3(
              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);
```

C.4 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 by means of 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
 by 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
 by the attributeId field, the list of possible values of attributeId is
 object-specific 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);
```

C.5 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 by means of 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
identifier 120. */

ret = sendAbcSecurityAlarmNotification(
       myNtfInstHandle,
       &name,
       “My Service User”,
       “My Service Provider”,
       “My Alarm Detector”,
       120,
       &ntfId);
```

C.6 Consumer Side (Example Function) – Subscribe for Notifications

```
/* Subscribe for all those kinds of notifications that are generated in the Producer API
examples by using the notification class identifiers as filter criteria.
*/

SaAisErrorT subscribeForAbcNotifications(SaNtfHandleT myNtfInstHandle)
{
       SaAisErrorT ret;

       SaNtfObjectCreateDeleteNotificationFilterT myOCDFilter;
       SaNtfAttributeChangeNotificationFilterT myAVCFilter;
       SaNtfStateChangeNotificationFilterT_2 mySCFilter;
       SaNtfAlarmNotificationFilterT myAFilter;
       SaNtfSecurityAlarmNotificationFilterT mySAFilter;

       SaNtfNotificationTypeFilterHandlesT_3 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_2(
       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 state id values */);

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_3(&abcNotificationFilterHandles, 1);

        saNtfNotificationFilterFree(myOCDFilter.notificationFilterHandle);
        saNtfNotificationFilterFree(myAVCFilter.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_3 *notification
)
{
        const SaNtfNotificationHeaderT *notificationHeader;
        SaNtfNotificationHandleT notificationHandle;
        void myNtfGenericHandler(
                        SaNtfNotificationHandleT notificationHandle,
                        const SaNtfNotificationHeaderT *notificationHeader);

        switch(notification->notificationType)
        {
        case SA_NTF_TYPE_OBJECT_CREATE_DELETE:
               notificationHeader =
               &(notification->notification.objectCreateDeleteNotification.notificationHeader);
               notificationHandle =
               notification->notification.objectCreateDeleteNotification.notificationHandle;
               break;
        case SA_NTF_TYPE_ATTRIBUTE_CHANGE:
               notificationHeader =
               &(notification->notification.attributeChangeNotification.notificationHeader);
               notificationHandle =
               notification->notification.attributeChangeNotification.notificationHandle;
               break;
        case SA_NTF_TYPE_STATE_CHANGE:
               notificationHeader =
               &(notification->notification.stateChangeNotification.notificationHeader);
               notificationHandle =
               notification->notification.stateChangeNotification.notificationHandle;
               break;
```

```
        case SA_NTF_TYPE_ALARM:
               notificationHeader =
               &(notification->notification.alarmNotification.notificationHeader);
               notificationHandle =
               notification->notification.alarmNotification.notificationHandle;
               break;
        case SA_NTF_TYPE_SECURITY_ALARM:
               notificationHeader =
               &(notification->notification.securityAlarmNotification.notificationHeader);
               notificationHandle =
               notification->notification.securityAlarmNotification.notificationHandle;
               break;
        }
        /* first do some generic notification handling */
        myNtfGenericHandler(notificationHandle, notificationHeader);

        /* then do some handling specific to 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(notificationHandle);
}


/* 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,
        const SaNtfNotificationHeaderT * header
)
```

```
{
        SaAisErrorT rc;
        SaStringT message = (SaStringT) NULL;
        char time_str[24];
        SaTimeT ntfTime = (SaTimeT)0;
        extern SaNtfHandleT ntfHandle;


        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);

        /* free resources; ntfHandle is defined globally */
        saNtfLocalizedMessageFree_2(ntfHandle, message);

}
```

An example for using the above functions:

```
SaAisErrorT ret;
SaVersionT version;
SaNtfHandleT ntfHandle,
SaNtfCallbacksT_3 ntfCallbacks;

 …

/* set up callback */
ntfCallbacks.saNtfNotificationCallback = myNotificationCallback;

/* then initialize the library instance */
ret = saNtfInitialize_3(&ntfHandle, &ntfCallbacks, &version);
if (ret != SA_AIS_OK)
{
       /* could not initialize the Notification Service library */
       exit (1);
}

/* subscribe for notifications */
ret = subscribeForAbcNotifications(ntfHandle);

…
```

C.7 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;
       SaNtfNotificationsT_3 returnedNotification;
       SaNtfNotificationTypeFilterHandlesT_3 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 */);

       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.notificationFilterHeader.notificationClassIds[0].majorId = 990;
       myAFilter.notificationFilterHeader.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_SEARCH_NOTIFICATION_ID;
criteria.notificationId = lastReceivedNotificationId;
ret = saNtfNotificationReadInitialize_3(&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_3(readHandle,SA_NTF_SEARCH_YOUNGER,
                                          &returnedNotification)) == SA_AIS_OK;)
{
        SaStringT destPtr = NULL;
        SaUint16T dataSize;
        SaNtfAlarmNotificationT *returnedANtf;

        returnedANtf = &returnedNotification.notification.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),
                        (void**) &destPtr,
                        &dataSize);
                … /* do something with the proposed repair action value
                        pointed to by destPtr */
        }
        saNtfNotificationFree(returnedANtf->notificationHandle);
}
```

```
        /* finalize reading */
        saNtfNotificationReadFinalize(readHandle);

        return ret;
}
```

Index of Definitions

A
additional information common parameter 30
additional text common parameter 30
alarm notifications 25
at-most-once delivery characteristic 38
attribute change notifications 26
attribute identifier attribute-change-notifications parameter 35
attribute identifier object-create/delete-notifications parameter 34
attribute identifier state-change-notifications parameter 33
attribute list object-create/delete-notifications parameter 34
attribute value object-create/delete-notifications parameter 34

C
cause security-alarm-notifications parameter 35
changed attribute list attribute-change-notifications parameter 35
changed state attribute list state-change-notifications parameter 33
cleared perceived severity alarm-notifications parameter 31
close-to-source suppression 42
completeness delivery characteristics 39
correlated notifications common parameter 30
critical perceived severity alarm-notifications parameter 32

D
delivery characteristics
  at-most-once delivery 38
  completeness 39
  guaranteed delivery 36
  ordering 38
  persistence 39
detector security-alarm-notifications parameter 36
discarded notifications 39
dynamic suppression 42

E
event time common parameter 30
event type common parameter 27

F
filtering
  see also notifications
  definition 40
  filters 25
  high level 40
  low level 40

G
guaranteed delivery characteristic 36

H
high level filtering 40

I
Indeterminate perceived severity alarm-notifications parameter 31

L
low level filtering 40

M
major perceived severity alarm-notifications parameter 32
minor perceived severity alarm-notifications parameter 32
miscellaneous notification 26
monitored attributes alarm-notifications parameter 32

N
NC *see* notification class common parameter
NCI *see* notification class identifier common parameter
new attribute value attribute-change-notifications parameter 35
new attribute value state-change-notifications parameter 33
notification 29
notification class common parameter 29
notification class identifier common parameter 29
notification filters 25
notification identifier common parameter 29
notification object common parameter 28
notification suppression *see* suppression
notification types 25
notifications
  see also filtering; notification-specific parameters; suppression
  definition 25
  common parameters
    additional information 30
    additional text 30
    correlated notifications 30
    event time 30
    event type 27
    notification class 29
    notification class identifier 29
    notification identifier 29
    notification object 28
    notifying object 28
  discarded 39
  types
    definition 25
    alarm 25
    attribute change 26
    miscellaneous 26
    object create/delete 26
    security alarm 26
    state change 26
notification-specific parameters
  see also notifications
  definition 31
  alarm
    monitored attributes 32
    perceived severity
      definition 31
      cleared 31
      critical 32
      indeterminate 31
      major 32

minor 32
warning 32
probable cause 31
proposed repair actions 32
specific problems 31
threshold information 32
trend indication 32
attribute change
changed attribute list
definition 35
attribute identifier 35
new attribute value 35
old attribute value 35
source indicator 35
object create/delete
attribute list
definition 34
attribute identifier 34
attribute value 34
source indicator 34
security alarm
cause 35
detector 36
service provider 36
service user 36
severity 35
state change
changed state attribute list
definition 33
attribute identifier 33
new attribute value 33
old attribute value 33
source indicator 33
notifying object common parameter 28

O
object create/delete notifications 26
old attribute value attribute-change-notifications parameter 35
old attribute value state-change-notifications parameter 33
ordering delivery characteristics 38

P
PDU-Readiness 46
perceived severity alarm-notifications parameter 31
persistence delivery characteristics 39
probable cause alarm-notifications parameter 31
producer API 25
proposed repair actions alarm-notifications parameter 32

R
reader API 25

S
security alarm notifications 26
service provider security-alarm-notifications parameter 36
service user security-alarm-notifications parameter 36
severity security-alarm-notifications parameter 35
source indicator attribute-change-notifications parameter 35
source indicator object-create/delete-notifications parameter 34
source indicator state-change-notifications parameter 33
specific problems alarm-notifications parameter 31
state change notifications 26
static suppression 42
subscriber API 25
suppression
see also notifications
definition 41
close-to-source 42
dynamic 42
static 42

T
threshold information alarm-notifications parameter 32
trend indication alarm-notifications parameter 32

W
warning perceived severity alarm-notifications parameter 32