# Service Availability™ Forum Application Interface Specification

Checkpoint Service SAI-AIS-CKPT-B.02.02


This specification was reissued on **September 30, 2011** under the Artistic License 2.0.
The technical contents and the version remain the same as in the original specification.

# 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 Checkpoint Service

# 1 Document Introduction

## 1.1 Document Purpose

This document defines the Checkpoint Service of the Application Interface Specification (AIS) of the Service Availability™ Forum (SA Forum). It is intended for use by implementers of the Application Interface Specification and by application developers who would use the Application Interface Specification to develop applications that must be highly available. 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 Interface Specification (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 Checkpoint Service specification:

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

(2) SAI-AIS-CKPT-B.01.01

(3) SAI-AIS-CKPT-B.02.01

This section presents the changes of the current release, SAI-AIS-CKPT-B.02.02, with respect to the SAI-AIS-CKPT-B.02.01 release. Editorial changes that do not change semantics or syntax of the described interfaces are not mentioned.

### 1.3.1 New Topics

- Chapter 4 contains the Checkpoint Service UML Information Model. This description was previously contained in the B.03.01 version of [1].
- Chapter 5 states that no administration APIs are provided for the Checkpoint Service in this release.
- Chapter 7 presents the Checkpoint Service management interface.

### 1.3.2 Clarifications

- The “section identifier descriptor” term has been introduced in Section 3.4.3.1 to distinguish its usage from the usage of the term “section identifier”. This issue has no impact on the APIs.
- The description of the `saCkptDispatch()` function (see Section 3.5.3) clarifies the meaning of the `SA_AIS_OK` return value.
- The description of the `saCkptFinalize()` function (see Section 3.5.4) explains the behavior of the Checkpoint Service if a process that still has an association with the Checkpoint Service terminates.
- The description of the `saCkptCheckpointClose()` function (see Section 3.6.3) clarifies that it frees all resources allocated by the Checkpoint Service for the invoking process for the specified checkpoint.

### 1.3.3 Changes in Return Values of API Functions

**Table 1 Changes in Return Values of API Functions**

| API Function | Return Value | Change Type |
|---|---|---|
| All API functions except `saCkptFinalize()` and all callback functions[1] | `SA_AIS_ERR_UNAVAILABLE` | new |
| `saCkptCheckpointOpen()`<br>`saCkptCheckpointOpenAsync()`<br>`SaCkptCheckpointOpenCallbackT` | `SA_AIS_ERR_NOT_EXIST`<br>`SA_AIS_ERR_EXIST` | clarified |
| `saCkptDispatch()` | `SA_AIS_OK` | clarified |

1. The `SaCkptCheckpointOpenCallbackT` and `SaCkptCheckpointSynchronizeCallbackT` callback functions have the `SA_AIS_ERR_UNAVAILABLE` return value in the `error` parameter.

### 1.3.4 Removed Topics

SA Forum revisited its alarm issuance directives and modified the conditions that determine when an alarm would be produced. As a consequence, AIS Services shall only generate alarms for situations that require an explicit intervention by an external agent or operator, provided that the corrective measures to be taken are well defined. Based on these directives, the alarms generated so far by the AIS Services have been revised, and it was decided to remove the "service impaired” alarm from the Checkpoint Service B.02.02 version.

SA Forum does not mandate that Checkpoint Service implementations which also support the B.02.01 version must generate the "service impaired" alarm for the B.02.01 version.

The "service impaired" alarm has also been removed from the Checkpoint Service MIB.

### 1.3.5 Other Changes

- In the description of the `saCkptInitialize()` function (see Section 3.5.1), the sentence
  "If the implementation supports the required `releaseCode`, and a major version >= the required `majorVersion`, `SA_AIS_OK` is returned."
  has been replaced by the sentence
  "If the implementation supports the specified `releaseCode` and `majorVersion`, `SA_AIS_OK` is returned.".
- The parameter named `id` in the prototype of the `saCkptSectionIdFree()` function (see Section 3.7.3) was mistakenly written as `Id` in the Checkpoint Service B.02.01 specification. This issue does not affect compatibility.
- In the description of the `saCkptSectionIterationNext()` function (see Section 3.7.6 of the B.02.02 release) of the B.02.01 Checkpoint Service specification, the reference
  "`sectionDescriptor->sectionId.id`" was mistakenly written
  "`sectionDescriptor->sectionId.->id`".

## 1.4 References

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

[1] Service Availability™ Forum, Service Availability Interface, Overview, SAI-Overview-B.04.01

[2] Service Availability™ Forum, Application Interface Specification, Notification Service, SAI-AIS-NTF-A.02.01

[3] Service Availability™ Forum, Application Interface Specification, Information Model Management Service, SAI-AIS-IMM-A.02.01

[4] Service Availability™ Forum, Application Interface Specification, Cluster Membership Service, SAI-AIS-CLM-B.03.01

[5] Service Availability™ Forum, SA Forum Information Model in XML Metadata Interchange (XMI) v2.1 format, SAI-XMI-A.03.01

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

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

## 1.5 How to Provide Feedback on the Specification

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

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

## 1.6 How to Join the Service Availability™ Forum

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

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

## 1.7 Additional Information

### 1.7.1 Member Companies

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

### 1.7.2 Press Materials

The Service Availability™ Forum has available a variety of downloadable resource materials, including the Forum Press Kit, graphics, and press contact information. Visit this area often for the latest press releases from the Service Availability™ Forum and its member companies by following the pertinent links provided on the Forum's website (http://www.saforum.org).

# 2 Overview

This specification defines the Checkpoint Service within the Application Interface Specification (AIS).

## 2.1 Checkpoint Service

The Checkpoint Service provides a facility for processes to record checkpoint data incrementally, which can be used to protect an application against failures. When recovering from fail-over or switch-over situations, the checkpoint data can be retrieved, and execution can be resumed from the state recorded before the failure.

Checkpoints are cluster-wide entities that are designated by unique names. A copy of the data stored in a checkpoint is called a checkpoint replica; for performance reasons, a checkpoint replica is typically stored in main memory rather than on disk. A checkpoint may have several checkpoint replicas stored on different nodes[1] in the cluster to protect it against node failures.

To avoid accumulation of unused checkpoints in the system, checkpoint replicas have a retention time. When a checkpoint has not been opened by any process for the duration of the retention time, the Checkpoint Service automatically deletes the checkpoint.

1. The term “node” without a qualifier is used in this document in the sense of a “member node”, as defined in the Cluster Membership Service specification (see [4]).

# 3 SA Checkpoint Service API

## 3.1 Checkpoint Service Model

### 3.1.1 Checkpoints

The Checkpoint Service manages a set of entities called **checkpoints**, which processes use to save their state. A process can use one or several checkpoints to save its state. Checkpoints are cluster-wide entities designated by unique names.

A process can dynamically create checkpoints and open existing checkpoints by invoking the `saCkptCheckpointOpen()` or the `saCkptCheckpointOpenAsync()` functions.
A process can close a checkpoint by invoking the `saCkptCheckpointClose()` function and can delete checkpoints by invoking the `saCkptCheckpointUnlink()` function. The interaction of the open, close, and unlink functions for checkpoints is similar to the way POSIX treats files and is described in detail in Section 3.6 on page 38.

To avoid the accumulation of unused checkpoints in the system, checkpoints have a **retention duration**. When a checkpoint has not been opened by any process for the retention duration, the Checkpoint Service automatically deletes the checkpoint.

If a process terminates abnormally, the Checkpoint Service automatically closes all of its open checkpoints.

### 3.1.2 Sections

Each checkpoint is structured to hold up to a maximum number of sections.
**Sections** contain raw data, which is not encoded by the Checkpoint Service. It is the responsibility of the process to encode the section contents if heterogeneity is a concern.

Within a checkpoint, each section is identified by a unique **section identifier**. Section identifiers are unique only within a checkpoint; sections in different checkpoints may have the same identifier. Section identifiers can be specified by the creating process or can be allocated dynamically by the Checkpoint Service.

The maximum number of sections in a checkpoint is specified when the checkpoint is created. For details, refer to the `saCkptCheckpointOpen()` and `saCkptCheckpointOpenAsync()` API functions. Sections belonging to a checkpoint can be dynamically created or deleted as long as the total number of sections does not exceed this maximum number. Within a checkpoint, sections of different

sizes may coexist. The size of a section can be changed dynamically as the result of the invocation of the `saCkptCheckpointWrite()` or `saCkptSectionOverwrite()` functions.

Sections have an **expiration time**, which is an absolute time. The Checkpoint Service automatically deletes a section when its expiration time is reached, regardless of whether the checkpoint is open by a process or not.
Note that the expiration time of a section is a time, in contrast to the retention duration of a checkpoint, which is a duration. Both expiration time and retention duration are defined as `SaTimeT`.

### 3.1.3 Checkpoint Replica

A copy of the data that is stored in a checkpoint is called a **checkpoint replica** or simply a **replica**. For a particular checkpoint, at most one checkpoint replica may reside on a particular node. A checkpoint may have several checkpoint replicas (or copies), each residing on another node.

A **local replica** is a replica located on the node where the checkpoint is opened.

The management of checkpoint replicas is for the most part transparent to the application programmer.

### 3.1.4 Checkpoint Data Access

A process can use the handle that the Checkpoint Service returned at open time to perform read and write operations on the checkpoint. A single read or write operation can access various portions of different sections within a checkpoint simultaneously.

Requirements regarding the consistency of the various replicas associated to a particular checkpoint can have negative effects on the performance of checkpoint write operations. To give flexibility to implementers of the Checkpoint Service, strong atomicity and strong ordering semantics are not required. In particular, if two processes perform a concurrent write to the same portion of a checkpoint, no global ordering of replica updates is ensured. After both write operations complete successfully, some replicas may contain data written by one process while other replicas contain data written by the other process. It is the responsibility of the processes to use proper synchronization mechanisms (such as the Lock Service) if such global update ordering is required.
However, ordering of updates issued by a single writer is required, even in the presence of faults. For example, assume that a thread within a process writes first `D1` in a section of a particular checkpoint and then writes `D2` in another section of the same checkpoint. Later on, if the same process or another process running on another node reads both sections entirely, and it sees `D2`, it must also see `D1`.

To accommodate different trade-offs between checkpoint update performance and replica consistency, different options are provided when a checkpoint is created. These options are described below.

### 3.1.5 Synchronous Update

When a checkpoint has been created with the **synchronous update** option, write and overwrite calls as well as calls for the creation and deletion of a section return only when all checkpoint replicas have been updated. In addition, the Checkpoint Service guarantees that a replica is not partially updated: a replica is either updated with all data specified in the write call or is not updated at all.
In other words, the Checkpoint Service guarantees that all replicas of a checkpoint created with the synchronous update option are identical if there are no concurrent overlapping updates to this checkpoint.

The Checkpoint Service does not specify from which replica checkpoint data is read.

A checkpoint with the synchronous update option can be created by specifying the `SA_CKPT_WR_ALL_REPLICAS` flag at creation time. For details, refer to Section 3.4.2.1 on page 23.

### 3.1.6 Asynchronous Update

For this update mode, the notion of an **active replica** is defined. It is a distinguished checkpoint replica whose properties are described below. At any time, at most one active replica exists.

When a checkpoint has been created with the **asynchronous update** option, write and overwrite calls as well as calls for the creation and deletion of a section return immediately when the active checkpoint replica has been updated. Other replicas are updated asynchronously. To guarantee that a process does not read stale data, the Checkpoint Service always reads from the active checkpoint replica.

If no active replica exists for a checkpoint created with the asynchronous update option, each of the section management operations of Section 3.7, except the `saCkptSectionIdFree()` function (see Section 3.7.3 on page 57), and each of the data access operations of Section 3.8, except the `saCkptIOVectorElementDataFree()` function (see Section 3.8.4 on page 74) will return an error.

The Checkpoint Service does not guarantee that all replicas of a checkpoint created with the asynchronous update option are always identical. However, a process can request that the Checkpoint Service **synchronizes** all checkpoint replicas by invoking the `saCkptCheckpointSynchronize()` or the

`saCkptCheckpointSynchronizeAsync()` functions to propagate checkpoint data to all of the checkpoint replicas.

This specification defines two variants of checkpoints with the asynchronous update option.
For the first one, the Checkpoint Service guarantees atomicity when replicas are updated, that is, a replica is either updated with all section data specified in the write call or is not updated at all. Such a checkpoint can be created by specifying the `SA_CKPT_WR_ACTIVE_REPLICA` flag at creation time. For details, refer t Section 3.4.2.1 on page 23.
The second variant does not provide the atomicity guarantee of the first one, but the Checkpoint Service marks sections that are modified by the write or overwrite calls as corrupt when a fault occurs while a checkpoint replica is being updated. Corrupted sections cannot be accessed by invoking `saCkptCheckpointRead()` or `saCkptCheckpointWrite()`; they can only be overwritten by invoking `saCkptSectionOverwrite()` or deleted by invoking `saCkptSectionDelete()`. Checkpoints with this **partial update** option are created by specifying the `SA_CKPT_WR_ACTIVE_REPLICA_WEAK` flag at creation time. For details, refer to Section 3.4.2.1 on page 23.

The partial update option is intended to be used by applications that may not want to pay the performance price associated with protection against partial updates.

**Note:** In the remainder of this document, if the term asynchronous update option is used without further specification, it refers to checkpoints created with any of the two variants of the asynchronous update option.

### 3.1.7 Management of Replicas for Collocated and Non-Collocated Checkpoints

#### *3.1.7.1 Collocated Checkpoints*

When using a checkpoint with the asynchronous update option, optimal performance for updating the checkpoint is achieved when the active replica is located on the same node as the process accessing the checkpoint. However, because the process accessing the checkpoint can change depending on the role assigned by the Availability Management Framework to it, optimal performance can be achieved only if the application informs the Checkpoint Service about which replica should be active at a particular time. An application informs the Checkpoint Service about the active replica for checkpoints having the collocated attribute. Such a checkpoint is named a **collocated checkpoint**. When a collocated checkpoint is created, no active replica exists until a local replica is set as the active replica by the `saCkptActiveReplicaSet()` call. An active replica stays active until the user explicitly sets another replica to active by calling the `saCkptActiveReplicaSet()` function, or if the replica is destroyed (for example if the node where this active replica resides crashes). In the

latter case, no active replica exists until the user sets a new one.
When `saCkptActiveReplicaSet()` returns, the Checkpoint Service guarantees that the new active replica is completely synchronized with the previous active replica. Data consistency for replica reads and writes and write ordering are preserved as though the change of active replica never took place. Replica reads or writes might be blocked until the synchronization completes.

The `saCkptActiveReplicaSet()` function can be used only for collocated checkpoints created with the asynchronous update option.

The management of replicas of collocated checkpoints and whether to set the replicas to active or not is mainly the duty of applications. Only applications can create a replica of a collocated checkpoint on a node. The replica is created by invoking one of the open calls on this node. If a replica for the checkpoint already exists on the node, the open call does not create another replica on this node.
The Checkpoint Service does not create replicas for collocated checkpoints other than the ones explicitly created by the applications by invoking an open call.

It is up to the applications to create enough number of replicas to have at any time the desired level of redundancy (for instance, by creating a replica on another node when a node becomes non-operational due to administrative operations).

#### *3.1.7.2 Non-Collocated Checkpoints*

Checkpoints created without the collocated attribute are called **non-collocated checkpoints**. The management of replicas of non-collocated checkpoints and whether to set the replicas to active or not (note that the notion “active” applies only to checkpoints created with the asynchronous update option) is mainly the duty of the Checkpoint Service.
The processes using the Checkpoint Service are not aware of the location of the active replicas of these checkpoints. The Checkpoint Service may create replicas other than the ones that may be created when opening a checkpoint. These other replicas can be useful to enhance the availability of checkpoints. For example, if two replicas exist at a certain point in time, and the node hosting one of these replicas is administratively taken out of service, the Checkpoint Service may allocate another replica on another node while this node is not available.

### 3.1.8 Persistence of Checkpoints

As has been stated in Section 2.1 on page 13, the Checkpoint Service typically stores checkpoint data in the main memory of the nodes. Regardless of the retention time, a checkpoint and all its sections do not survive if the Checkpoint Service stops running on all nodes hosting replicas for this checkpoint. The stop of the Checkpoint Service can be caused by administrative actions or node failures.

## 3.2 Unavailability of the Checkpoint Service API on a Non-Member Node

The Checkpoint 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 Checkpoint Service under various conditions that cause the Checkpoint Service to be unavailable on a cluster node. Section 3.2.2 contains guidelines for Checkpoint Service implementers for dealing with a temporary unavailability of the service.

### 3.2.1 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 Checkpoint Service behaves as follows towards processes residing on that cluster node and using or attempting to use the service:

⇒ Calls to `saCkptInitialize()` will fail with SA_AIS_ERR_UNAVAILABLE.

⇒ All Checkpoint 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 following exceptions, assuming that the handle `ckptHandle` has already been acquired, and no other errors occur:

- The `saCkptCheckpointOpenAsync()` function may return SA_AIS_OK or SA_AIS_ERR_UNAVAILABLE, depending on the service implementation. If it returns SA_AIS_OK, the callback `SaCkptCheckpointOpenCallbackT` will be called and will also return SA_AIS_ERR_UNAVAILABLE in the `error` parameter; otherwise, the callback will not be called.
- The `saCkptCheckpointSynchronizeAsync()` function may return SA_AIS_OK or SA_AIS_ERR_UNAVAILABLE, depending on the service implementation. If it returns SA_AIS_OK, the callback `SaCkptCheckpointSynchronizeCallbackT` will be called and will also return SA_AIS_ERR_UNAVAILABLE in the `error` parameter; otherwise, the callback will not be called.
- The `saCkptFinalize()` function, which is used to free the library handles and all resources associated with these handles, will return SA_AIS_OK.

⇒ Outstanding callbacks `SaCkptCheckpointOpenCallbackT` and `SaCkptCheckpointSynchronizeCallbackT` will return SA_AIS_ERR_UNAVAILABLE in the `error` parameter.

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 Checkpoint 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 (or with the special treatment described above for asynchronous calls) with the exception of saCkptFinalize(), 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 Checkpoint Service executing on the remaining nodes of the cluster behaves as if all processes that were using the Checkpoint Service on the leaving cluster node had been terminated. In particular, if an saCkptCheckpointUnlink() operation is pending because one or more processes on the leaving cluster node had the checkpoint open, the unlink operation can proceed now.

### 3.2.2 Guidelines for Checkpoint Service Implementers

The implementation of the Checkpoint 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 3.2.1 before returning SA_AIS_ERR_UNAVAILABLE. If the Cluster Membership Service considers a cluster node as a member of the cluster but the Checkpoint 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.

## 3.3 Include File and Library Names

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

```
#include <saCkpt.h>
```

To use the Checkpoint Service API, an application must be bound with the following library:

```
libSaCkpt.so
```

## 3.4 Type Definitions

The Checkpoint Service uses the types described in the following sections.

### 3.4.1 Handles

#### *3.4.1.1 SaCkptHandleT*

```
typedef SaUint64T SaCkptHandleT;
```

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

#### *3.4.1.2 SaCkptCheckpointHandleT*

```
typedef SaUint64T SaCkptCheckpointHandleT;
```

The type of the handle of a checkpoint.

#### *3.4.1.3 SaCkptSectionIterationHandleT*

```
typedef SaUint64T SaCkptSectionIterationHandleT;
```

The type of a handle for stepping through the sections in a checkpoint.

## 3.4.2 Checkpoint Types

### *3.4.2.1 SaCkptCheckpointCreationFlagsT*

```
#define SA_CKPT_WR_ALL_REPLICAS            0X1
#define SA_CKPT_WR_ACTIVE_REPLICA          0X2
#define SA_CKPT_WR_ACTIVE_REPLICA_WEAK     0X4
#define SA_CKPT_CHECKPOINT_COLLOCATED      0X8
typedef SaUint32T SaCkptCheckpointCreationFlagsT;
```

The flags of the `SaCkptCheckpointCreationFlagsT` type have the following interpretation:

- `SA_CKPT_WR_ALL_REPLICAS` - The specification of this flag at creation time creates a checkpoint with the synchronous update option. Any of the operations that modify the checkpoint (`saCkptSectionWrite()`, `saCkptSectionOverwrite()`, `saCkptSectionCreate()`, and `saCkptSectionDelete()`) will be performed on all of the checkpoint replicas before the operation returns. It also guarantees atomicity for each checkpoint replica, that is, either the invocation succeeds, or it fails, and nothing has been written to any of the replicas.
- `SA_CKPT_WR_ACTIVE_REPLICA` - The specification of this flag at creation time creates a checkpoint with the asynchronous update option, which supports atomicity when replicas are updated: any of the operations that modify the checkpoint will be performed on the active checkpoint replica before the operation returns. The atomicity when updating replicas means: for each of the checkpoint replicas, either the operation on the replica succeeds, or it fails, and nothing has been written to the replica.
- `SA_CKPT_WR_ACTIVE_REPLICA_WEAK` - The specification of this flag at creation time creates a checkpoint with the partial update option. Any of the operations that modify the checkpoint will be performed on the active checkpoint replica before the call returns. However, no guarantee of atomicity per checkpoint replica is provided, that is, if the operation of writing does not complete successfully, some sections might get corrupted.
- `SA_CKPT_CHECKPOINT_COLLOCATED` - A checkpoint created with such attribute is called a collocated checkpoint; otherwise, it is called a non-collocated checkpoint. For details, refer to Section 3.1.7 on page 18.

The flags `SA_CKPT_WR_ALL_REPLICAS`, `SA_CKPT_WR_ACTIVE_REPLICA`, and `SA_CKPT_WR_ACTIVE_REPLICA_WEAK` are mutually exclusive. A value of type `SaCkptCheckpointCreationFlagsT` is either one of the mutually exclusive flags

or the bitwise OR of one of these mutually exclusive flags and the `SA_CKPT_CHECKPOINT_COLLOCATED` flag.

#### *3.4.2.2 SaCkptCheckpointCreationAttributesT*

```
typedef struct {
      SaCkptCheckpointCreationFlagsT creationFlags;
      SaSizeT checkpointSize;
      SaTimeT retentionDuration;
      SaUint32T maxSections;
      SaSizeT maxSectionSize;
      SaSizeT maxSectionIdSize;
} SaCkptCheckpointCreationAttributesT;
```

The attributes defined in the `SaCkptCheckpointCreationAttributesT` structure are as follows:

- `creationFlags` - These flags specify the collocation attribute of checkpoint replicas and the update option (synchronous or asynchronous). For details, refer to Section 3.4.2.1 on page 23.
- `retentionDuration` - The duration for which the checkpoint will be retained while it is not opened by any process. The `retentionDuration` starts after the last checkpoint user has closed the checkpoint.
- `checkpointSize` - The net size in bytes of each checkpoint replica that can be used for application data.
- `maxSections` - The maximum number of sections in the checkpoint. Every checkpoint has at least one section. If and only if `maxSections` is 1, a default section is automatically created. This default section is identified by the special section identifier descriptor `SA_CKPT_DEFAULT_SECTION_ID` (see Section 3.4.3.1 on page 25).
- `maxSectionSize` - The upper bound on the possible size of the sections in this checkpoint.
- `maxSectionIdSize` - The maximum length of the section identifier in the checkpoint. To be accepted as a valid parameter, this value must not equal zero and may be required to be greater than an implementation-specific minimal value.

**Note:**

- `checkpointSize` =< `maxSections` * `maxSectionSize`
- `checkpointSize` does not include the space needed for storing `maxSections` * `maxSectionIdSize`.

#### *3.4.2.3 SaCkptCheckpointOpenFlagsT*

```
#define SA_CKPT_CHECKPOINT_READ       0X1
#define SA_CKPT_CHECKPOINT_WRITE      0X2
#define SA_CKPT_CHECKPOINT_CREATE     0X4
typedef SaUint32T SaCkptCheckpointOpenFlagsT;
```

The `SaCkptCheckpointOpenFlagsT` type has the following interpretation:

- `SA_CKPT_CHECKPOINT_READ` - The checkpoint is opened in read mode.
- `SA_CKPT_CHECKPOINT_WRITE` - The checkpoint is opened in write mode. In order to modify either data or metadata associated with a checkpoint, the checkpoint must be opened with this flag.
- `SA_CKPT_CHECKPOINT_CREATE` - The checkpoint is created if it does not already exist.

### 3.4.3 Section Types

#### *3.4.3.1 SaCkptSectionIdT*

```
#define SA_CKPT_DEFAULT_SECTION_ID          {0, NULL}
#define SA_CKPT_GENERATED_SECTION_ID        {0, NULL}
```

These special constants define the **section identifier descriptor** of the default section and the section identifier descriptor of a generated section. The section identifier descriptor is represented by the `SaCkptSectionIdT` structure:

```
typedef struct {
     SaUint16T idLen;
     SaUint8T *id;
} SaCkptSectionIdT;
```

The fields of the `SaCkptSectionIdT` structure are the length of the **section identifier** and a pointer to the section identifier.

#### *3.4.3.2 SaCkptSectionCreationAttributesT*

```
typedef struct {
     SaCkptSectionIdT *sectionId;
     SaTimeT expirationTime;
} SaCkptSectionCreationAttributesT;
```

The fields of the `SaCkptSectionCreationAttributesT` structure have the following interpretation:

- `sectionId` - [in/out] Pointer to the section identifier descriptor that identifies the section to be created. If the section identifier descriptor has the special value `SA_CKPT_GENERATED_SECTION_ID`, the Checkpoint Service automatically generates a new section identifier and changes the values of the fields in the section identifier descriptor.
- `expirationTime` - [in] The absolute time after which the Checkpoint Service deletes the section automatically. The `expirationTime` can be specified when a section is created, or it can be later modified by invoking `saCkptSectionExpirationTimeSet()`. If `expirationTime` has the special value `SA_TIME_END`, the Checkpoint Service never deletes the section automatically.

#### *3.4.3.3 SaCkptSectionStateT*

```
typedef enum {
     SA_CKPT_SECTION_VALID           = 1,
     SA_CKPT_SECTION_CORRUPTED       = 2
} SaCkptSectionStateT;
```

The values of the `SaCkptSectionStateT` enumeration type indicate that the section is either valid or corrupted.

#### *3.4.3.4 SaCkptSectionDescriptorT*

```
typedef struct {
     SaCkptSectionIdT sectionId;
     SaTimeT expirationTime;
     SaSizeT sectionSize;
     SaCkptSectionStateT sectionState;
     SaTimeT lastUpdate;
} SaCkptSectionDescriptorT;
```

The fields of the `SaCkptSectionDescriptorT` structure have the following interpretation:

- `sectionId` - The section identifier descriptor that identifies the section.
- `expirationTime` - The absolute time at which the section will be deleted.
- `sectionSize` - The amount of the section that is currently in use.
- `sectionState` - The state of the section. A section is either in the `SA_CKPT_SECTION_VALID` state or the `SA_CKPT_SECTION_CORRUPTED` state. A section can be in the `SA_CKPT_SECTION_CORRUPTED` state when the checkpoint has been created with the `SA_CKPT_WR_ACTIVE_REPLICA_WEAK` property, and an invocation of `saCkptCheckpointWrite()` or `saCkptSectionOverwrite()` did not complete successfully.
- `lastUpdate` - The absolute time of the last update.

#### *3.4.3.5 SaCkptSectionsChosenT*

```
typedef enum {
     SA_CKPT_SECTIONS_FOREVER                  = 1,
     SA_CKPT_SECTIONS_LEQ_EXPIRATION_TIME      = 2,
     SA_CKPT_SECTIONS_GEQ_EXPIRATION_TIME      = 3,
     SA_CKPT_SECTIONS_CORRUPTED                = 4,
     SA_CKPT_SECTIONS_ANY                      = 5
} SaCkptSectionsChosenT;
```

The values of the `SaCkptSectionChosenT` enumeration type are used for searching existing sections fulfilling the specified criteria. The values have the following interpretation:

- SA_CKPT_SECTIONS_FOREVER - All sections with expiration time set to SA_TIME_END.
- SA_CKPT_SECTIONS_LEQ_EXPIRATION_TIME - All sections with expiration time less than or equal to the value of expirationTime.
- SA_CKPT_SECTIONS_GEQ_EXPIRATION_TIME - All sections with expiration time greater than or equal to the value of expirationTime.
- SA_CKPT_SECTIONS_CORRUPTED - All corrupted sections.
- SA_CKPT_SECTIONS_ANY - All sections.

### 3.4.4 IoVector Types

#### *3.4.4.1 SaCkptIOVectorElementT*

```
typedef struct {
     SaCkptSectionIdT sectionId;
     void *dataBuffer;
     SaSizeT dataSize;
     SaOffsetT dataOffset;
     SaSizeT readSize;
} SaCkptIOVectorElementT;
```

The fields of the SaCkptIOVectorElementT structure have the following interpretation:

- sectionId - The section identifier descriptor that identifies the section into which it is to be written or from which it is to be read.
- dataBuffer - A pointer to a buffer that contains data to be written to the section or data read from the section.
- dataSize - Either the size in bytes of the data to be written to the section from the buffer pointed to by dataBuffer or the size in bytes of the data to be read from the section into the buffer pointed to by dataBuffer. This size is at most maxSectionSize, as specified in the creation attributes of the checkpoint.
- dataOffset - Offset in the section that marks the start of the data to be written to it or to be read from it.
- readSize - Used by saCkptCheckpointRead() to record the number of bytes of data that have been read from the section; otherwise, this field is ignored.

### 3.4.5 SaCkptCheckpointDescriptorT

```
typedef struct {
     SaCkptCheckpointCreationAttributesT
          checkpointCreationAttributes;
     SaUint32T numberOfSections;
     SaSizeT memoryUsed;
} SaCkptCheckpointDescriptorT;
```

The fields of the `SaCkptCheckpointDescriptorT` structure have the following interpretation:

- `checkpointCreationAttributes` - Structure containing the checkpoint attributes that were set when the checkpoint was created by an invocation of the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions.
- `numberOfSections` - The number of sections in the checkpoint.
- `memoryUsed` - The number of bytes used in the checkpoint to store checkpoint data.

### 3.4.6 SaCkptCallbacksT

The `SaCkptCallbacksT` structure is defined as follows:

```
typedef struct {
     SaCkptCheckpointOpenCallbackT
          saCkptCheckpointOpenCallback;
     SaCkptCheckpointSynchronizeCallbackT
          saCkptCheckpointSynchronizeCallback;
} SaCkptCallbacksT;
```

The callbacks structure is supplied to the Checkpoint Service by a process and contains the callback functions that the Checkpoint Service can invoke.

### 3.4.7 SaCkptCheckpointStatusT

The following enum describes the various states of a checkpoint that can be notified to a system administrator in form of a **state change** notification. For details on notifications, refer to Chapter 6 and [2].

```
typedef enum {
     SA_CKPT_SECTION_RESOURCES_EXHAUSTED= 1,
     SA_CKPT_SECTION_RESOURCES_AVAILABLE= 2
} SaCkptCheckpointStatusT;
```

The values of the `SaCkptCheckpointStatusT` enumeration type have the following interpretation:

- `SA_CKPT_SECTION_RESOURCES_EXHAUSTED` - The maximum number of sections is reached, or no memory is left for write access.
- `SA_CKPT_SECTION_RESOURCES_AVAILABLE` - Memory is available for write access, or at least one section within the checkpoint is available, after having recovered from a preceding `SA_CKPT_SECTION_RESOURCES_EXHAUSTED` condition.

### 3.4.8 SaCkptStateT

The following enum holds all the checkpoint state types. Currently, only one such state is defined:

```
typedef enum {
     SA_CKPT_CHECKPOINT_STATUS = 1
} SaCkptStateT;
```

## 3.5 Library Life Cycle

### 3.5.1 saCkptInitialize()

#### Prototype

```
SaAisErrorT saCkptInitialize(
      SaCkptHandleT *ckptHandle,
      const SaCkptCallbacksT *ckptCallbacks,
      SaVersionT *version
);
```

#### Parameters

`ckptHandle` - [`out`] A pointer to the handle which identifies this particular initialization of the Checkpoint Service, and which is to be returned by the Checkpoint Service. The `SaCkptHandleT` type is defined in Section 3.4.1.1 on page 22.

`ckptCallbacks` - [`in`] If `ckptCallbacks` is set to NULL, no callback is registered; If `ckptCallbacks` is not set to NULL, it is a pointer to an `SaCkptCallbacksT` structure which contains the callback functions of the process that the Checkpoint Service may invoke. Only non-NULL callback functions in this structure will be registered. The `SaCkptCallbacksT` type is defined in Section 3.4.6 on page 29.

`version` - [`in/out`] As an input parameter, `version` is a pointer to a structure containing the required Checkpoint 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 Checkpoint Service. The `SaVersionT` type is defined in [1].

#### Description

This function initializes the Checkpoint Service for the invoking process and registers the various callback functions. This function must be invoked prior to the invocation of any other Checkpoint Service functionality. The handle pointed to by `ckptHandle` is returned by the Checkpoint Service as the reference to this association between the process and the Checkpoint Service. The process uses this handle in subsequent communication with the Checkpoint Service.

If the implementation supports the specified `releaseCode` and `majorVersion`, `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 can support for the required `releaseCode`
- `minorVersion` = highest value of the minor version that this implementation can support for the required value of `releaseCode` and the returned value of `majorVersion`

If the 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 Checkpoint Service library or the Checkpoint Service provider 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 Checkpoint 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**

`saCkptSelectionObjectGet()`, `saCkptDispatch()`, `saCkptFinalize()`

### 3.5.2 saCkptSelectionObjectGet()

**Prototype**

```
SaAisErrorT saCkptSelectionObjectGet(
      SaCkptHandleT ckptHandle,
      SaSelectionObjectT *selectionObject
);
```

**Parameters**

`ckptHandle` - [in] The handle which was obtained by a previous invocation of the `saCkptInitialize()` function and which identifies this particular initialization of the Checkpoint Service. The `SaCkptHandleT` type is defined in Section 3.4.1.1 on page 22.

`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 [1].

**Description**

This function returns the operating system handle associated with the handle `ckptHandle`. The invoking process can use the operating system handle to detect pending callbacks, instead of repeatedly invoking the `saCkptDispatch()` function 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 `saCkptSelectionObjectGet()` is valid until `saCkptFinalize()` is invoked on the same handle `ckptHandle`.

### 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 `ckptHandle` 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 Checkpoint Service library or the Checkpoint Service provider 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 `ckptHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptInitialize()`, `saCkptDispatch()`, `saCkptFinalize()`

### 3.5.3 saCkptDispatch()

#### Prototype

```
SaAisErrorT saCkptDispatch(
      SaCkptHandleT ckptHandle,
      SaDispatchFlagsT dispatchFlags
);
```

#### Parameters

`ckptHandle` - [in] The handle which was obtained by a previous invocation of the `saCkptInitialize()` function and which identifies this particular initialization of the Checkpoint Service. The `SaCkptHandleT` type is defined in Section 3.4.1.1 on page 22.

`dispatchFlags` - [in] Flags that specify the callback execution behavior of the `saCkptDispatch()` function, which 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 [1].

#### Description

In the context of the calling thread, this function invokes pending callbacks for the handle `ckptHandle` in the way 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 `ckptHandle` 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 `ckptHandle` 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 ckptHandle was acquired before the cluster node left the cluster membership.

**See Also**

saCkptInitialize(), saCkptSelectionObjectGet()

### 3.5.4 saCkptFinalize()

**Prototype**

```
SaAisErrorT saCkptFinalize(
     SaCkptHandleT ckptHandle
);
```

**Parameters**

ckptHandle - [in] The handle which was obtained by a previous invocation of the saCkptInitialize() function and which identifies this particular initialization of the Checkpoint Service. The SaCkptHandleT type is defined in Section 3.4.1.1 on page 22.

**Description**

The saCkptFinalize() function closes the association represented by the ckptHandle parameter between the invoking process and the Checkpoint Service. The process must have invoked saCkptInitialize() before it invokes this function. A process must invoke this function once for each handle acquired by invoking saCkptInitialize().

If the saCkptFinalize() function completes successfully, it releases all resources acquired when saCkptInitialize() was called. Moreover, it closes all checkpoints that are open for the particular handle. Furthermore, it cancels all pending SaCkptCheckpointOpenCallbackT 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 Checkpoint Service implicitly finalizes all instances of the Checkpoint Service that are associated with the process, as described in the preceding paragraph.

After `saCkptFinalize()` completes successfully, the handle `ckptHandle` 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 `ckptHandle` is invalid, since it is corrupted, uninitialized, or has already been finalized.

### See Also

`saCkptInitialize()`

## 3.6 Checkpoint Management

### 3.6.1 saCkptCheckpointOpen() and saCkptCheckpointOpenAsync()

#### Prototype

```
SaAisErrorT saCkptCheckpointOpen(
      SaCkptHandleT ckptHandle,
      const SaNameT *checkpointName,
      const SaCkptCheckpointCreationAttributesT
           *checkpointCreationAttributes,
      SaCkptCheckpointOpenFlagsT checkpointOpenFlags,
      SaTimeT timeout,
      SaCkptCheckpointHandleT *checkpointHandle
);

SaAisErrorT saCkptCheckpointOpenAsync(
      SaCkptHandleT ckptHandle,
      SaInvocationT invocation,
      const SaNameT *checkpointName,
      const SaCkptCheckpointCreationAttributesT
           *checkpointCreationAttributes,
      SaCkptCheckpointOpenFlagsT checkpointOpenFlags
);
```

#### Parameters

`ckptHandle` - [`in`] The handle which was obtained by a previous invocation of the `saCkptInitialize()` function and which identifies this particular initialization of the Checkpoint Service. The `SaCkptHandleT` type is defined in Section 3.4.1.1 on page 22.

`invocation` - [`in`] A parameter that identifies a particular invocation of the response callback. The `SaInvocationT` type is defined in [1].

`checkpointName` - [`in`] A pointer to the checkpoint name that identifies a checkpoint globally in a cluster. The `SaNameT` type is defined in [1].

`checkpointCreationAttributes` - [in] A pointer to the creation attributes of a checkpoint. The `SaCkptCheckpointCreationAttributesT` type is defined in Section 3.4.2.2 on page 24.
If the user intends only to open an existing checkpoint, `checkpointCreationAttributes` must be set to NULL and the `SA_CKPT_CHECKPOINT_CREATE` flag in `checkpointOpenFlags` may not be set. If the user intends to open a checkpoint or create and open a checkpoint if it does not exist, the structure to which `checkpointCreationAttributes` points must contain the attributes for the checkpoint, and the `SA_CKPT_CHECKPOINT_CREATE` flag in `checkpointOpenFlags` must be set. If the checkpoint already exists, it is not recreated, and the call succeeds only if the creation attributes match the ones used at creation time, excluding `retentionDuration`, which is ignored, as it may be independently modified by invoking the `saCkptCheckpointRetentionDurationSet()` function.

`checkpointOpenFlags` - [in] The value of this parameter is constructed by a bitwise OR of the flags defined by the `SaCkptCheckpointOpenFlagsT` type in Section 3.4.2.3 on page 25.

`timeout` - [in] The `saCkptCheckpointOpen()` invocation is considered to have failed if it does not complete within the time specified. It is unspecified whether a checkpoint is created or not. The `SaTimeT` type is defined in [1].

`checkpointHandle` - [out] A pointer to the checkpoint handle, allocated in the address space of the invoking process. If the checkpoint is opened successfully, the Checkpoint Service stores into the memory area to which `checkpointHandle` points the handle that the process uses to access the checkpoint in subsequent invocations of the functions of the Checkpoint Service API. In the case of `saCkptCheckpointOpenAsync()`, this handle is returned in the corresponding callback. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

### Description

The `saCkptCheckpointOpen()` and `saCkptCheckpointOpenAsync()` functions open a checkpoint. If the checkpoint does not exist and the `SA_CKPT_CHECKPOINT_CREATE` flag is set in the `checkpointOpenFlags` parameter, the checkpoint is first created.

An invocation of `saCkptCheckpointOpen()` is blocking. A new checkpoint handle is returned upon successful completion. A checkpoint can be opened multiple times for reading or writing in the same or different processes.

When a checkpoint replica is created as a result of this invocation, the following is guaranteed:

- If the checkpoint has been created with the SA_CKPT_WR_ALL_REPLICAS flag set, the checkpoint replica must be identical to the other checkpoint replicas.
- Otherwise, the data in this new checkpoint replica is synchronized using the data in the active checkpoint replica.

The completion of the saCkptCheckpointOpenAsync() function is signaled by the associated saCkptCheckpointOpenCallback() callback function, which must have been supplied when the process invoked the saCkptInitialize() call. The process supplies the value of invocation when it invokes the saCkptCheckpointOpenAsync() function and the Checkpoint Service gives that value of invocation back to the application when it invokes the corresponding saCkptCheckpointOpenCallback() function. The invocation parameter is a mechanism that enables the process to determine which call triggered which callback.

### 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, or the timeout, specified by the timeout parameter, 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 ckptHandle is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INIT - The previous invocation of saCkptInitialize() to initialize the Checkpoint Service was incomplete, since the saCkptCheckpointOpenCallback() callback function is missing. This return value only applies to the saCkptCheckpointOpenAsync() function.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly. In particular, this error is returned for each of the following cases:

- The user specifies checkpointCreationAttributes with checkpointSize > maxSections * maxSectionSize.
- The SA_CKPT_CHECKPOINT_CREATE flag is not set, and checkpointCreationAttributes is not NULL.
- The SA_CKPT_CHECKPOINT_CREATE flag is set and checkpointCreationAttributes is NULL.
- The SA_CKPT_CHECKPOINT_CREATE flag is set, and the name to which checkpointName points is not a valid checkpoint DN.

SA_AIS_ERR_NO_MEMORY - Either the Checkpoint Service library or the Checkpoint Service provider 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 SA_CKPT_CHECKPOINT_CREATE flag is not set, the checkpointCreationAttributes is NULL, and the checkpoint designated by the name to which checkpointName points does not exist.

SA_AIS_ERR_EXIST - The SA_CKPT_CHECKPOINT_CREATE flag is set, the checkpoint already exists, and the creation attributes in the structure to which checkpointCreationAttributes points (excluding retentionDuration, which is ignored) are different from the ones used at creation time.

SA_AIS_ERR_BAD_FLAGS - The checkpointOpenFlags 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 ckptHandle was acquired before the cluster node left the cluster membership.

### See Also

SaCkptCheckpointOpenCallbackT, saCkptCheckpointClose(), saCkptInitialize(), saCkptCheckpointRetentionDurationSet()

### 3.6.2 SaCkptCheckpointOpenCallbackT

#### Prototype

```
typedef void (*SaCkptCheckpointOpenCallbackT)(
      SaInvocationT invocation,
      SaCkptCheckpointHandleT checkpointHandle,
      SaAisErrorT error
);
```

#### Parameters

`invocation` - [in] This parameter was supplied by a process in the corresponding invocation of the `saCkptCheckpointOpenAsync()` function and is used by the Checkpoint Service in this callback. This invocation parameter allows the process to match the invocation of that function with this callback. The `SaInvocationT` type is defined in [1].

`checkpointHandle` - [in] The handle that identifies the checkpoint. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`error` - [in] This parameter indicates whether the `saCkptCheckpointOpenAsync()` function was successful. The `SaAisErrorT` type is defined in [1]. The returned values are:

- `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 try again.
- `SA_AIS_ERR_BAD_HANDLE` - The handle `ckptHandle` in the corresponding invocation of the `saCkptCheckpointOpenAsync()` function is invalid, since it is corrupted, uninitialized, or has already been finalized.
- `SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly in the corresponding invocation of the `saCkptCheckpointOpenAsync()` function. In particular, this error is returned for each of the following cases:

  - The user specifies `checkpointCreationAttributes` with `checkpointSize` > `maxSections` * `maxSectionSize`.
  - The `SA_CKPT_CHECKPOINT_CREATE` flag is not set, and `checkpointCreationAttributes` is not NULL.
  - The `SA_CKPT_CHECKPOINT_CREATE` flag is set and `checkpointCreationAttributes` is NULL.
  - The `SA_CKPT_CHECKPOINT_CREATE` flag is set, and the name to which `checkpointName` points is not a valid checkpoint DN.
- `SA_AIS_ERR_NO_MEMORY` - Either the Checkpoint Service library or the Checkpoint Service provider 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` - In the corresponding invocation of the `saCkptCheckpointOpenAsync()` function, the `SA_CKPT_CHECKPOINT_CREATE` flag is not set, the `checkpointCreationAttributes` is NULL, and the checkpoint designated by the name to which `checkpointName` points does not exist.
- `SA_AIS_ERR_EXIST` - In the corresponding invocation of the `saCkptCheckpointOpenAsync()` function, the `SA_CKPT_CHECKPOINT_CREATE` flag is set, the checkpoint already exists, and the creation attributes in the structure to which `checkpointCreationAttributes` points (excluding `retentionDuration`, which is ignored) are different from the ones used at creation time.
- `SA_AIS_ERR_BAD_FLAGS` - The `checkpointOpenFlags` parameter in the corresponding invocation of the `saCkptCheckpointOpenAsync()` function is invalid.
- `SA_AIS_ERR_UNAVAILABLE` - The operation requested in the corresponding invocation of the `saCkptCheckpointOpenAsync()` function 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 `ckptHandle` specified in the corresponding invocation of the `saCkptCheckpointOpenAsync()` function was acquired before the cluster node left the cluster membership.

#### Description

The Checkpoint Service calls this callback function when the operation requested by the invocation of `saCkptCheckpointOpenAsync()` completes.

This callback is invoked in the context of a thread calling `saCkptDispatch()` on the handle `ckptHandle` that was specified in the `saCkptCheckpointOpenAsync()` call.

If successful, the reference to the opened/created checkpoint is returned in `checkpointHandle`; otherwise, an error is returned in the `error` parameter.

#### Return Values

None

#### See Also

`saCkptCheckpointOpenAsync()`, `saCkptDispatch()`, `saCkptCheckpointClose()`

### 3.6.3 saCkptCheckpointClose()

#### Prototype

```
SaAisErrorT saCkptCheckpointClose(
      SaCkptCheckpointHandleT checkpointHandle
);
```

#### Parameters

`checkpointHandle` - [in] The handle that identifies the checkpoint to close. The handle `checkpointHandle` must have been obtained previously by the invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

#### Description

This API function closes the checkpoint identified by `checkpointHandle` and which was opened by an earlier invocation of either `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`.

After this invocation, the handle `checkpointHandle` is no longer valid.

This call frees all resources allocated for this process by the Checkpoint Service for the checkpoint identified by the handle `checkpointHandle`. In particular, this call

frees memory allocated for the process and which has not yet been freed by the `saCkptSectionIdFree()` or `saCkptIOVectorElementDataFree()` functions.

This call cancels all pending callbacks that refer directly or indirectly to the handle `checkpointHandle`. Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

When the invocation of the `saCkptCheckpointClose()` function completes successfully, if no process has the checkpoint open any longer, the following will occur:

- The checkpoint is deleted immediately if its deletion was pending as a result of an `saCkptCheckpointUnlink()` function, or
- the checkpoint will be deleted when the retention duration expires if no process opens it in the meantime.

The deletion (unlink) of a checkpoint frees all resources allocated by the Checkpoint Service for it.

When a process terminates, all of its opened checkpoints are closed.

**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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

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 checkpointHandle was acquired before the cluster node left the cluster membership.

### See Also

saCkptCheckpointOpen(), saCkptCheckpointOpenAsync(), SaCkptCheckpointOpenCallbackT, saCkptCheckpointUnlink()

## 3.6.4 saCkptCheckpointUnlink()

### Prototype

```
SaAisErrorT saCkptCheckpointUnlink(
     SaCkptHandleT ckptHandle,
     const SaNameT *checkpointName
);
```

### Parameters

ckptHandle - [in] The handle which was obtained by a previous invocation of the saCkptInitialize() function and which identifies this particular initialization of the Checkpoint Service. The SaCkptHandleT type is defined in Section 3.4.1.1 on page 22.

checkpointName - [in] A pointer to the name of the checkpoint to be unlinked. The SaNameT type is defined in [1].

### Description

This function deletes from the cluster an existing checkpoint identified by the name to which checkpointName points.

After successful completion of the invocation, the following applies:

- The name to which checkpointName points is no longer valid, that is, any invocation of a function of the Checkpoint Service API that uses this checkpoint name returns an error unless a checkpoint is re-created with this name. The checkpoint is re-created by specifying in an open call the SA_CKPT_CHECKPOINT_CREATE flag and the same name of the checkpoint to

be unlinked. This way, a new instance of the checkpoint is created while the old instance of the checkpoint is possibly not yet finally deleted.
Note that this behavior is similar to the way POSIX treats files.

- If no process has the checkpoint open when `saCkptCheckpointUnlink()` is invoked, the checkpoint is immediately deleted.
- Any process that has the checkpoint open can still continue to access it. Deletion of the checkpoint will occur when the last `saCkptCheckpointClose()` operation is performed.

The deletion of a checkpoint frees all resources allocated by the Checkpoint Service for it.

This API can be invoked by any process, and the invoking process need not be the creator or opener of the checkpoint.

### 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 `ckptHandle` 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_NOT_EXIST` - The checkpoint identified by the name to which `checkpointName` points does not exist.

`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 `ckptHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptCheckpointClose()`

### 3.6.5 saCkptCheckpointRetentionDurationSet()

#### Prototype

```
SaAisErrorT saCkptCheckpointRetentionDurationSet(
      SaCkptCheckpointHandleT checkpointHandle,
      SaTimeT retentionDuration
);
```

#### Parameters

`checkpointHandle` - [in] The handle that identifies the checkpoint whose retention time is being set. The handle `checkpointHandle` must have been obtained previously by the invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`retentionDuration` - [in] The value of the retention duration to be set. The checkpoint is retained (not deleted) for the retention duration. The `SaNameT` type is defined in [1].

#### Description

The function `saCkptCheckpointRetentionDurationSet()` sets the retention duration of the checkpoint identified by `checkpointHandle` to `retentionDuration`. If the last checkpoint user closes the checkpoint, and the checkpoint is not opened by any process within the retention duration, the Checkpoint Service automatically deletes the checkpoint.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

#### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_ACCESS - The checkpoint identified by `checkpointHandle` was not opened for write mode.

SA_AIS_ERR_BAD_OPERATION - The retention duration of the checkpoint identified by `checkpointHandle` cannot be changed as the checkpoint has been unlinked.

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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

**See Also**

`saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `SaCkptCheckpointOpenCallbackT`, `saCkptCheckpointClose()`, `saCkptCheckpointUnlink()`

### 3.6.6 saCkptActiveReplicaSet()

**Prototype**

```
SaAisErrorT saCkptActiveReplicaSet(
     SaCkptCheckpointHandleT checkpointHandle
);
```

**Parameters**

`checkpointHandle` - [in] The handle that identifies a checkpoint. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

### Description

This function can be used only for checkpoints that have been created with the collocated attribute and the asynchronous update option.

The local checkpoint replica will become the active replica when this function completes successfully.

A local replica that was set active by a previous call to `saCkptActiveReplicaSet()` and was not overridden by another call to `saCkptActiveReplicaSet()` on another node, remains active until the checkpoint expires, or the replica is destroyed.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions has already been finalized.

`SA_AIS_ERR_ACCESS` - The checkpoint identified by `checkpointHandle` was not opened for write mode.

`SA_AIS_ERR_BAD_OPERATION` - The checkpoint identified by `checkpointHandle` was not created as a collocated checkpoint with the asynchronous update option.

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 checkpointHandle was acquired before the cluster node left the cluster membership.

#### See Also

saCkptCheckpointWrite(), saCkptSectionOverwrite(), saCkptCheckpointRead(), saCkptCheckpointOpen(), saCkptCheckpointOpenAsync(), SaCkptCheckpointOpenCallbackT

### 3.6.7 saCkptCheckpointStatusGet()

#### Prototype

```
SaAisErrorT saCkptCheckpointStatusGet(
      SaCkptCheckpointHandleT checkpointHandle,
      SaCkptCheckpointDescriptorT *checkpointStatus
);
```

#### Parameters

checkpointHandle - [in] The handle to the checkpoint whose status is to be returned. The handle checkpointHandle must have been obtained previously by the invocation of saCkptCheckpointOpen() or saCkptCheckpointOpenCallback(). The SaCkptCheckpointHandleT type is defined in Section 3.4.1.2 on page 22.

checkpointStatus - [out] A pointer to an SaCkptCheckpointDescriptorT structure (allocated in the address space of the invoking process) into which this function returns the checkpoint status information. The SaCkptCheckpointDescriptorT structure is defined in Section 3.4.5 on page 29.

#### Description

This function retrieves the checkpoint status of the checkpoint identified by checkpointHandle and writes the checkpoint status into the structure to which checkpointStatus points.

If the checkpoint was created with either the SA_CKPT_WR_ACTIVE_REPLICA option or the SA_CKPT_WR_ACTIVE_REPLICA_WEAK option, the checkpoint status is

obtained from the active replica. If the checkpoint was created with the SA_CKPT_WR_ALL_REPLICAS option, the Checkpoint Service determines the replica from which to obtain the checkpoint status.

This function will fail with the SA_AIS_ERR_ACCESS return value if the checkpoint identified by `checkpointHandle` was not opened for read mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NOT_EXIST - No active replica exists.

SA_AIS_ERR_ACCESS - The checkpoint identified by `checkpointHandle` was not opened for read mode.

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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `SaCkptCheckpointOpenCallbackT`

## 3.7 Section Management

### 3.7.1 saCkptSectionCreate()

#### Prototype

```
SaAisErrorT saCkptSectionCreate(
     SaCkptCheckpointHandleT checkpointHandle,
     SaCkptSectionCreationAttributesT
          *sectionCreationAttributes,
     const void *initialData,
     SaSizeT initialDataSize
);
```

#### Parameters

`checkpointHandle` - [in] The handle to the checkpoint that is to hold the section to be created. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`sectionCreationAttributes` - [in/out] A pointer to an `SaCkptSectionCreationAttributesT` structure (as defined in Section 3.4.3.2 on page 26) that contains the `in/out` field `sectionId` and the `in` field `expirationTime`. The `sectionId` field is a pointer to the section identifier descriptor, which identifies the section to be created. If the section identifier descriptor has the special value `SA_CKPT_GENERATED_SECTION_ID`, the Checkpoint Service automatically generates a new section identifier and changes the values of the fields in the section identifier descriptor (`SaCkptSectionIdT` structure). In this case, the `id` field points to an area of memory that is allocated by the Checkpoint Service library to contain the new section identifier. The invoking process must free this memory by calling the `saCkptSectionIdFree()` function.

`initialData` - [in] Pointer to a location (in the address space of the invoking process) that contains the initial data of the section to be created.

`initialDataSize` - [in] The size in bytes of the initial data of the section to be created. This size can be at most `maxSectionSize`, as specified by the checkpoint creation attributes in the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions. The `SaSizeT` type is defined in [1].

### Description

This function creates a new section in the checkpoint identified by `checkpointHandle`, provided that the total number of existing sections is less than the maximum number of sections specified in one of the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` API calls.
Unlike a checkpoint, a section does not need to be opened for access.
The Checkpoint Service deletes the section when its expiration time is reached.
If a checkpoint is created to have only one section, it is not necessary to create that section.
The default section is identified by the special section identifier descriptor `SA_CKPT_DEFAULT_SECTION_ID`.
If the checkpoint was created with the `SA_CKPT_WR_ALL_REPLICAS` property, the section is created in all of the checkpoint replicas when the invocation returns; otherwise, the section has been created at least in the active checkpoint replica when the invocation returns and will be created asynchronously in the other checkpoint replicas.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

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

`SA_AIS_ERR_NO_RESOURCES` - The system does not have enough resources to create this section.

`SA_AIS_ERR_NO_SPACE` - With the creation of this new section, the maximum number of sections specified for this checkpoint would be exceeded. The section is not created.

`SA_AIS_ERR_NOT_EXIST` - No active replica exists.

`SA_AIS_ERR_ACCESS` - The checkpoint identified by `checkpointHandle` was not opened for write mode.

`SA_AIS_ERR_EXIST` - The section identified by the section identifier descriptor referred to by `sectionId` in the structure to which `sectionCreationAttributes` points already exists, or the checkpoint was created to have only one section.

`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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptSectionDelete()`, `saCkptSectionIdFree()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `SaCkptCheckpointOpenCallbackT`

### 3.7.2 saCkptSectionDelete()

#### Prototype

```
SaAisErrorT saCkptSectionDelete(
      SaCkptCheckpointHandleT checkpointHandle,
      const SaCkptSectionIdT *sectionId
);
```

#### Parameters

`checkpointHandle` - [in] The handle to the checkpoint holding the section to be deleted. The handle `checkpointHandle` must have been obtained previously by the invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`sectionId` - [in] A pointer to the section identifier descriptor of the section to be deleted. The `SaCkptSectionIdT` type is defined in Section 3.4.3.1 on page 25.

#### Description

This function deletes a section in the checkpoint identified by `checkpointHandle`. If the checkpoint was created with the `SA_CKPT_WR_ALL_REPLICAS` property, the section has been deleted in all of the checkpoint replicas when the invocation returns; otherwise, the section has been deleted at least in the active checkpoint replica when the invocation returns. The default section identified by the value `SA_CKPT_DEFAULT_SECTION_ID` of the section identifier descriptor cannot be deleted by invoking the `saCkptSectionDelete()` function.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

#### 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 checkpointHandle is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the saCkptCheckpointOpen() or saCkptCheckpointOpenCallback() functions, or the corresponding checkpoint has already been closed.
- The handle ckptHandle that was passed to the functions saCkptCheckpointOpen() or saCkptCheckpointOpenAsync() has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly. In particular, this value is returned if the caller attempts to delete the default section.

SA_AIS_ERR_NOT_EXIST - No active replica exists, or the section identified by the section identifier descriptor to which sectionId points does not exist.

SA_AIS_ERR_ACCESS - The checkpoint identified by checkpointHandle was not opened for write mode.

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 checkpointHandle was acquired before the cluster node left the cluster membership.

**See Also**

saCkptSectionCreate(), saCkptCheckpointOpen(), saCkptCheckpointOpenAsync(), SaCkptCheckpointOpenCallbackT

### 3.7.3 saCkptSectionIdFree()

**Prototype**

```
SaAisErrorT saCkptSectionIdFree(
     SaCkptCheckpointHandleT checkpointHandle,
     SaUint8T *id
);
```

**Parameters**

checkpointHandle - [in] The handle to the checkpoint. The handle checkpointHandle must have been obtained by a previous invocation of

`saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`id` - [`in`] A pointer to the section identifier that was allocated by the Checkpoint Service library in the `saCkptSectionCreate()` function and is to be freed. The `SaUint8T` type is defined in [1].

### Description

This function frees the memory to which `id` points. This memory was allocated by the Checkpoint Service library in a previous call to the `saCkptSectionCreate()` function.

For details, refer to the description of the `sectionCreationAttributes` parameter in the corresponding invocation of the `saCkptSectionCreate()` function.

### 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_BAD_HANDLE` - The handle `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptSectionCreate()`

### 3.7.4 saCkptSectionExpirationTimeSet()

#### Prototype

```
SaAisErrorT saCkptSectionExpirationTimeSet(
      SaCkptCheckpointHandleT checkpointHandle,
      const SaCkptSectionIdT* sectionId,
      SaTimeT expirationTime
);
```

#### Parameters

`checkpointHandle` - [`in`] The handle to the checkpoint that contains the section for which the expiration time is to be set. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`sectionId` - [`in`] A pointer to the section identifier descriptor of the section for which the expiration time is to be set. The `SaCkptSectionIdT` type is defined in Section 3.4.3.1 on page 25.

`expirationTime` - [`in`] The expiration time that is to be set for the section identified by the section identifier descriptor to which `sectionId` points. The expiration time is an absolute time that defines the time at which the Checkpoint Service will delete the section automatically, regardless of whether the checkpoint is open by a process or not.
If `expirationTime` has the special value `SA_TIME_END`, the Checkpoint Service never deletes the section automatically. The `SaTimeT` type is defined in [1].

#### Description

This function sets the expiration time of the section identified by the section identifier descriptor pointed to by `sectionId` within the checkpoint identified by the handle `checkpointHandle` to the value `expirationTime`. The expiration time of the default section identified by `SA_CKPT_DEFAULT_SECTION_ID` is unlimited and cannot be changed.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly. In particular, this value is returned if the section identifier descriptor of the section to which `sectionId` points is set to SA_CKPT_DEFAULT_SECTION_ID.

SA_AIS_ERR_NOT_EXIST - No active replica exists, or the section identified by the section identifier descriptor to which `sectionId` points does not exist.

SA_AIS_ERR_ACCESS - The checkpoint identified by `checkpointHandle` was not opened for write mode.

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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptSectionCreate()`, `saCkptCheckpointOpen()`, `SaCkptCheckpointOpenCallbackT`

### 3.7.5 saCkptSectionIterationInitialize()

#### Prototype

```
SaAisErrorT saCkptSectionIterationInitialize(
      SaCkptCheckpointHandleT checkpointHandle,
      SaCkptSectionsChosenT sectionsChosen,
      SaTimeT expirationTime,
      SaCkptSectionIterationHandleT *sectionIterationHandle
);
```

#### Parameters

`checkpointHandle` - [in] The checkpoint handle which was obtained previously by an invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`sectionsChosen` - [in] A predicate (as defined by the `SaCkptSectionsChosenT` structure in Section 3.4.3.5 on page 27) that describes the sections to be chosen during an iteration.

`expirationTime` - [in] An absolute time used by `sectionsChosen`, as described above. This field is ignored when `sectionsChosen` is `SA_CKPT_SECTIONS_FOREVER`, `SA_CKPT_SECTIONS_CORRUPTED`, or `SA_CKPT_SECTIONS_ANY`. The `SaTimeT` type is defined in [1].

`sectionIterationHandle` - [out] A pointer to a memory area allocated by the invoking process to contain the section iteration handle. If this function returns successfully, the Checkpoint Service stores into this memory area the handle that the process uses in subsequent invocations of the `saCkptSectionIterationNext()` and `saCkptSectionIterationFinalize()` functions for stepping through the sections of the checkpoint identified by `checkpointHandle`. The `SaCkptSectionIterationHandleT` type is defined in Section 3.4.1.3 on page 22.

#### Description

This function returns the section iteration handle for stepping through the sections of a checkpoint identified by `checkpointHandle`. The iteration steps only through sections that match the criteria specified in `sectionsChosen`. The Checkpoint Service keeps track of the current position while iterating through sections.

This function will fail with the SA_AIS_ERR_ACCESS return value if the checkpoint identified by `checkpointHandle` was not opened for read mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NOT_EXIST - No active replica exists. This return code applies only to checkpoints created with the asynchronous update option.

SA_AIS_ERR_ACCESS - The checkpoint identified by `checkpointHandle` was not opened for read mode.

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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptSectionIterationNext()`, `saCkptSectionIterationFinalize()`, `saCkptSectionCreate()`, `saCkptSectionDelete()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`

### 3.7.6 saCkptSectionIterationNext()

#### Prototype

```
SaAisErrorT saCkptSectionIterationNext(
     SaCkptSectionIterationHandleT sectionIterationHandle,
     SaCkptSectionDescriptorT *sectionDescriptor
);
```

#### Parameters

`sectionIterationHandle` - [in] The section iteration handle which was obtained by an invocation of the `saCkptSectionIterationInitialize()` function for stepping through the sections of a checkpoint. The `SaCkptSectionIterationHandleT` type is defined in Section 3.4.1.3 on page 22.

`sectionDescriptor` - [out] A pointer to an `SaCkptSectionDescriptorT` structure (as defined in Section 3.4.3.4 on page 27), which is allocated by the invoking process. The `saCkptSectionIterationNext()` function places the requested information into this structure. Note that the memory to which `sectionDescriptor->sectionId.id` points is allocated by the Checkpoint Service, and it is up to the Checkpoint Service to free this memory either

- at the next invocation of the `saCkptSectionIterationNext()` to continue the iteration, or
- when the `saCkptSectionIterationFinalize()` function is called to finalize the iteration, or
- when the corresponding checkpoint is closed, or
- the handle of this particular initialization of the Checkpoint Service is finalized.

The memory area to which `sectionDescriptor` points should be deallocated by the invoking process.

#### Description

This function iterates over an internal table of sections by using the handle `sectionIterationHandle` which was obtained by a previous invocation of the `saCkptSectionIterationInitialize()` function. When the function returns successfully, the structure to which `sectionDescriptor` points is set to the descriptor of a section. A subsequent invocation of `saCkptSectionIterationNext()` returns another section. When there are no more sections to return, an error is returned.

Every section created before the invocation of the `saCkptSectionIterationInitialize()` function and not deleted before the invocation of `saCkptSectionIterationFinalize()` will be returned exactly once by this invocation. No other guarantees are given; sections that are created after an iteration is initialized or deleted before an iteration is finalized, may or may not be returned by an invocation of this function.

### 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 `sectionIterationHandle` is invalid, due to one or more of the reasons below:

- The handle `sectionIterationHandle` is either corrupted, or it was not obtained with the `saCkptSectionIterationInitialize()` function, or `saCkptSectionIterationFinalize()` has already been invoked.
- The checkpoint identified by `checkpointHandle`, which was specified in the corresponding `saCkptSectionIterationInitialize()` call, has already been closed.
- The handle `ckptHandle` that was passed to the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

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

`SA_AIS_ERR_NOT_EXIST` - No active replica exists. This return code applies only to checkpoints created with the asynchronous update option.

`SA_AIS_ERR_NO_SECTIONS` - There are no more sections matching `sectionsChosen`.

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 sectionIterationHandle was acquired before the cluster node left the cluster membership.

#### See Also

saCkptSectionIterationInitialize(), saCkptSectionIterationFinalize(), saCkptSectionCreate(), saCkptSectionDelete(), saCkptCheckpointOpen(), saCkptCheckpointOpenAsync()

### 3.7.7 saCkptSectionIterationFinalize()

#### Prototype

```
SaAisErrorT saCkptSectionIterationFinalize(
     SaCkptSectionIterationHandleT sectionIterationHandle
);
```

#### Parameters

sectionIterationHandle - [in] The section iteration handle which was obtained by an invocation of the saCkptSectionIterationInitialize() function and which identifies the iteration to be finalized. The SaCkptSectionIterationHandleT type is defined in Section 3.4.1.3 on page 22.

#### Description

This function frees resources allocated for iteration.

#### 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 `sectionIterationHandle` is invalid, due to one or more of the reasons below:

- The handle `sectionIterationHandle` is either corrupted, or it was not obtained with the `saCkptSectionIterationInitialize()` function, or `saCkptSectionIterationFinalize()` has already been invoked.
- The checkpoint identified by `checkpointHandle`, which was specified in the corresponding `saCkptSectionIterationInitialize()` call, has already been closed.
- The handle `ckptHandle` that was passed to the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` functions has already been finalized.

SA_AIS_ERR_NOT_EXIST - No active replica exists. This return code applies only to checkpoints created with the asynchronous update option.

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 `sectionIterationHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptSectionIterationInitialize()`, `saCkptSectionIterationNext()`, `saCkptSectionCreate()`, `saCkptSectionDelete()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`

## 3.8 Data Access

### 3.8.1 saCkptCheckpointWrite()

#### Prototype

```
SaAisErrorT saCkptCheckpointWrite(
      SaCkptCheckpointHandleT checkpointHandle,
      const SaCkptIOVectorElementT *ioVector,
      SaUint32T numberOfElements,
      SaUint32T *erroneousVectorIndex
);
```

#### Parameters

`checkpointHandle` - [in] The handle to the checkpoint into which data is to be written. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`ioVector` - [in] A pointer to an array with elements `ioVector`[0], ..., `ioVector`[`numberOfElements` - 1]. The type of each element is `SaCkptIOVectorElementT`, as defined in Section 3.4.4.1 on page 28. Each element contains the following fields: `sectionId`, `dataBuffer`, `dataSize`, `dataOffset`, and `readSize`. If `sectionId` is equal to `SA_CKPT_DEFAULT_SECTION_ID`, the default section is written. The value of `dataSize` is at most `maxSectionSize`, as specified in the creation attributes of the checkpoint. The field `readSize` is ignored by the `saCkptCheckpointWrite()` function.

`numberOfElements` - [in] The number of elements in the array to which the `ioVector` parameter points. The `SaUint32T` type is defined in [1].

`erroneousVectorIndex` - [in/out] This parameter is ignored if the `saCkptCheckpointWrite()` function succeeds. Otherwise, and it is not NULL, it specifies a pointer to a field (in the address space of the invoking process) where the Checkpoint Service places the index of the first element of the array pointed to by the `ioVector` parameter at which the invocation fails. The index zero corresponds to `ioVector`[0]. The `SaUint32T` type is defined in [1].

### Description

This function writes data from the memory area(s) specified by the `ioVector` parameter into a checkpoint:

- If the checkpoint has been created with the `SA_CKPT_WR_ALL_REPLICAS` property, all of the checkpoint replicas have been updated when the invocation returns `SA_AIS_OK`; otherwise, nothing has been written at all.
- If the checkpoint has been created with the `SA_CKPT_WR_ACTIVE_REPLICA` property, the active checkpoint replica has been updated when the invocation returns `SA_AIS_OK`. Other checkpoint replicas are updated asynchronously. If the invocation does not return `SA_AIS_OK`, nothing has been written at all.
- If the checkpoint been created with the `SA_CKPT_WR_ACTIVE_REPLICA_WEAK` property, the active checkpoint replica has been updated when the invocation returns `SA_AIS_OK`. Other checkpoint replicas are updated asynchronously. If the invocation returns with an error, nothing has been written at all. However, if the invocation does not complete because the process exited while calling `saCkptCheckpointWrite()`, the operation may be partially completed, and some sections may be corrupted in the active checkpoint replica.

In a single invocation, several sections and several portions of sections can be updated simultaneously. The elements of the array to which the `ioVector` parameter points are written in order from `ioVector`[0] to `ioVector`[`numberOfElements` - 1]. As a result of this invocation, some sections might grow.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Checkpoint Service library or the Checkpoint Service provider 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 active replica exists, or a section identified by the section identifier descriptor `sectionId` in an element of the array to which the `ioVector` parameter points does not exist.

SA_AIS_ERR_ACCESS - The checkpoint identified by `checkpointHandle` was not opened for write mode.

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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptSectionOverwrite()`, `saCkptCheckpointRead()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `SaCkptCheckpointOpenCallbackT`

### 3.8.2 saCkptSectionOverwrite()

#### Prototype

```
SaAisErrorT saCkptSectionOverwrite(
      SaCkptCheckpointHandleT checkpointHandle,
      const SaCkptSectionIdT *sectionId,
      const void *dataBuffer,
      SaSizeT dataSize
);
```

#### Parameters

`checkpointHandle` - [in] The handle that identifies the checkpoint containing a section to be overwritten. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`sectionId` - [in] A pointer to the section identifier descriptor of the section to overwrite. If the section identifier descriptor has the value `SA_CKPT_DEFAULT_SECTION_ID`, the default section is updated. The `SaCkptSectionIdT` type is defined in Section 3.4.3.1 on page 25.

`dataBuffer` - [in] A pointer to a buffer that contains the data to be written.

`dataSize` - [in] The size in bytes of the data to be written. This value becomes the new size for this section. The `SaSizeT` type is defined in [1].

#### Description

This function is similar to `saCkptCheckpointWrite()`, but it overwrites only a single section. As a result of this invocation, the previous data and size of the section may change. This function may be invoked for a section, even if the `saCkptCheckpointWrite()` function was not previously invoked for the section.

This function will fail with the `SA_AIS_ERR_ACCESS` return value if the checkpoint identified by `checkpointHandle` was not opened for write mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Checkpoint Service library or the Checkpoint Service provider 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 active replica exists, or the section identified by the section identifier descriptor to which `sectionId` points does not exist.

SA_AIS_ERR_ACCESS - The checkpoint identified by `checkpointHandle` was not opened for write mode.

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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptCheckpointRead()`, `saCkptCheckpointWrite()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `SaCkptCheckpointOpenCallbackT`

## 3.8.3 saCkptCheckpointRead()

### Prototype

```
SaAisErrorT saCkptCheckpointRead(
      SaCkptCheckpointHandleT checkpointHandle,
      SaCkptIOVectorElementT *ioVector,
      SaUint32T numberOfElements,
      SaUint32T *erroneousVectorIndex
);
```

### Parameters

`checkpointHandle` - [`in`] The handle to the checkpoint from which data is to be read. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`ioVector` - [`in`/`out`] A pointer to an array that contains elements `ioVector`[0], ..., `ioVector`[`numberOfElements` - 1]. The type of each element is `SaCkptIOVectorElementT`, as defined in Section 3.4.4.1 on page 28. An element contains the following fields:

- `sectionId` - [`in`] The section identifier descriptor of the section from which to read.
- `dataBuffer` - [`in`/`out`] A pointer to a buffer into which read section data is placed. If `dataBuffer` is NULL, the value of `dataSize` provided by the invoker is ignored and the buffer is provided by the Checkpoint Service library. The buffer must be freed by the invoking process by calling the `saCkptIOVectorElementDataFree()` function. Providing a NULL `dataBuffer` indicates also that the intention is to read from `dataOffset` up to the end of the section identified by the section identifier descriptor `sectionId`.
- `dataSize` - [`in`] Size of the data to be read from the section to the buffer to which `dataBuffer` points. The size is at most `maxSectionSize`, as specified in the creation attributes of the checkpoint.

- `dataOffset` - [in] Offset in the section that marks the start of the data to be read from the section.
- `readSize` - [out] Used by `saCkptCheckpointRead()` to record the number of bytes of data that have been read from the section.

`numberOfElements` - [in] The number of elements in the array to which the `ioVector` parameter points. The `SaUint32T` type is defined in [1].

`erroneousVectorIndex` - [in/out] This parameter is ignored if the `saCkptCheckpointRead()` function succeeds. Otherwise, and it is not NULL, it specifies a pointer to a field (in the address space of the invoking process) where the Checkpoint Service places the index of the first element of the array pointed to by `ioVector` parameter at which the invocation fails. The index zero corresponds to the element `ioVector`[0]. The `SaUint32T` type is defined in [1].

### Description

This function copies data from a checkpoint replica into the array specified by `ioVector`. Some of the buffers provided to the invocation may have been modified if the invocation does not succeed.

When the buffer to which `dataBuffer` points is allocated by the Checkpoint Service library, care must be taken to ensure that the invoking process deallocates that buffer space promptly.

This function will fail with the SA_AIS_ERR_ACCESS return value if the checkpoint identified by `checkpointHandle` was not opened for read mode.

### 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 `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback)` functions, or the corresponding checkpoint has already been closed.

- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

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

`SA_AIS_ERR_NOT_EXIST` - No active replica exists, or a section identified by the section identifier descriptor `sectionId` in one of the elements of the array to which `ioVector` points does not exist.

`SA_AIS_ERR_ACCESS` - The checkpoint identified by `checkpointHandle` was not opened for read mode.

`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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

**See Also**

`saCkptIOVectorElementDataFree()`, `saCkptCheckpointWrite()`, `saCkptSectionOverwrite()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `SaCkptCheckpointOpenCallbackT`

### 3.8.4 saCkptIOVectorElementDataFree()

**Prototype**

```
SaAisErrorT saCkptIOVectorElementDataFree(
     SaCkptCheckpointHandleT checkpointHandle,
     void *data
);
```

**Parameters**

`checkpointHandle` - [in] The handle of the checkpoint. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`data` - [in] A pointer to the buffer that was allocated by the Checkpoint Service library in the `saCkptCheckpointRead()` function and is to be freed.

### Description

This function frees the memory to which `data` points and which was allocated by the Checkpoint Service library in a previous call to the `saCkptCheckpointRead()` function.

For details, refer to the description of the `ioVector` parameter in the corresponding invocation of the `saCkptCheckpointRead()` function.

### 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_BAD_HANDLE` - The handle `checkpointHandle` is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback)` functions, or the corresponding checkpoint has already been closed.
- The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`saCkptCheckpointRead()`

### 3.8.5 saCkptCheckpointSynchronize(), saCkptCheckpointSynchronizeAsync()

#### Prototype

```
SaAisErrorT saCkptCheckpointSynchronize(
      SaCkptCheckpointHandleT checkpointHandle,
      SaTimeT timeout
);

SaAisErrorT saCkptCheckpointSynchronizeAsync(
      SaCkptCheckpointHandleT checkpointHandle,
      SaInvocationT invocation
);
```

#### Parameters

`checkpointHandle` - [in] The handle of the checkpoint to be synchronized. The handle `checkpointHandle` must have been obtained by a previous invocation of `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()`. The `SaCkptCheckpointHandleT` type is defined in Section 3.4.1.2 on page 22.

`invocation` - [in] This parameter identifies a particular invocation of the response callback. The `SaInvocationT` type is defined in [1].

`timeout` - [in] The function will terminate if the time it takes exceeds `timeout`; however, the propagation of the checkpoint data to other checkpoint replicas might continue even if this error is returned. The `SaTimeT` type is defined in [1].

#### Description

The `saCkptCheckpointSynchronize()` and `saCkptCheckpointSynchronizeAsync()` functions ensure that all previous operations applied on the active checkpoint replica are propagated to other checkpoint replicas. Such operations are `saCkptCheckpointWrite()`, `saCkptSectionOverwrite()`, `saCkptSectionCreate()`, and `saCkptSectionDelete()`.

It is not guaranteed that new operations applied while the synchronization is in progress will be propagated when the synchronization operation completes. If new operations are applied while the synchronization is in progress, it is not guaranteed that the replicas are all identical when `saCkptCheckpointSynchronize()` returns, or the `saCkptCheckpointSynchronizeCallback()` callback is invoked

(see the description of the flags SA_CKPT_WR_ACTIVE_REPLICA and SA_CKPT_WR_ACTIVE_REPLICA_WEAK in Section 3.4.2.1 on page 23).

The saCkptCheckpointSynchronize() and saCkptCheckpointSynchronizeAsync() functions apply only to checkpoints created with the asynchronous update option.

If the specified timeout expires when invoking the saCkptCheckpointSynchronize() function, it is not guaranteed that the checkpoint replicas have been synchronized.

The completion of the saCkptCheckpointSynchronizeAsync() function is signaled by the associated saCkptCheckpointSynchronizeCallback() callback function, which must have been supplied when the process invoked the saCkptInitialize() call.
The invoking process sets the invocation parameter and the Checkpoint Service uses the value of invocation in the invocation of the callback function.

The SA_AIS_ERR_ACCESS value is returned if the checkpoint identified by checkpointHandle was not opened for write mode.

**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, or the timeout specified by the timeout parameter 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 checkpointHandle is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained with the saCkptCheckpointOpen() or saCkptCheckpointOpenCallback() functions, or the corresponding checkpoint has already been closed.
- The handle ckptHandle that was passed to the functions saCkptCheckpointOpen() or saCkptCheckpointOpenAsync() has already been finalized.

`SA_AIS_ERR_INIT` - The previous invocation of `saCkptInitialize()` to initialize the Checkpoint Service was incomplete, since the `saCkptCheckpointSynchronizeCallback()` callback function is missing. This return value applies only to the `saCkptCheckpointSynchronizeAsync()` function.

`SA_AIS_ERR_INVALID_PARAM` - A parameter is not set correctly.

`SA_AIS_ERR_NO_MEMORY` - Either the Checkpoint Service library or the Checkpoint Service provider 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 active replica exists.

`SA_AIS_ERR_ACCESS` - The checkpoint identified by `checkpointHandle` was not opened for write mode.

`SA_AIS_ERR_BAD_OPERATION` - The checkpoint identified by `checkpointHandle` was not created with the asynchronous update option.

`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 `checkpointHandle` was acquired before the cluster node left the cluster membership.

### See Also

`SaCkptCheckpointSynchronizeCallbackT`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `saCkptInitialize()`, `saCkptCheckpointWrite()`, `saCkptSectionOverwrite()`, `saCkptSectionCreate()`, `saCkptSectionDelete()`, `SaCkptCheckpointOpenCallbackT`

### 3.8.6 SaCkptCheckpointSynchronizeCallbackT

#### Prototype

```
typedef void (*SaCkptCheckpointSynchronizeCallbackT)(
     SaInvocationT invocation,
     SaAisErrorT error
);
```

#### Parameters

`invocation` - [in] This parameter is supplied by a process in the corresponding invocation of the `saCkptCheckpointSynchronize()` function and is used by the Checkpoint Service in this callback. This invocation parameter allows the process to match the invocation of that function with this callback. The `SaInvocationT` type is defined in [1].

`error` - [in] This parameter indicates whether the `saCkptCheckpointSynchronize()` function was successful. The `SaAisErrorT` type is defined in [1]. The possible return values are:

- `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 `checkpointHandle` that was passed to the corresponding `saCkptCheckpointSynchronizeAsync()` function is invalid, due to one or both of the reasons below:
  - It is corrupted, was not obtained with the `saCkptCheckpointOpen()` or `saCkptCheckpointOpenCallback()` functions, or the corresponding checkpoint has already been closed.
  - The handle `ckptHandle` that was passed to the functions `saCkptCheckpointOpen()` or `saCkptCheckpointOpenAsync()` has already been finalized.

- SA_AIS_ERR_INVALID_PARAM - A parameter was not set correctly in the corresponding invocation of the saCkptCheckpointSynchronizeAsync() function.
- SA_AIS_ERR_NO_MEMORY - Either the Checkpoint Service library or the Checkpoint Service provider 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 active replica exists.
- SA_AIS_ERR_ACCESS - The checkpoint identified by checkpointHandle in the corresponding invocation of the saCkptCheckpointSynchronizeAsync() function was not opened for write mode.
- SA_AIS_ERR_BAD_OPERATION - The checkpoint identified by checkpointHandle in the corresponding invocation of the saCkptCheckpointSynchronizeAsync() function was not created with the asynchronous update option.
- SA_AIS_ERR_UNAVAILABLE - The operation requested in the corresponding invocation of the saCkptCheckpointSynchronizeAsync() function 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 checkpointHandle specified in the corresponding invocation of the saCkptCheckpointSynchronizeAsync() function was acquired before the cluster node left the cluster membership.

### Description

The Checkpoint Service invokes this callback function when the operation requested by the invocation of saCkptCheckpointSynchronizeAsync() completes.

This callback is invoked in the context of a thread calling saCkptDispatch() on the handle ckptHandle that is associated with the saCkptCheckpointSynchronizeAsync() call. Associated means here that ckptHandle was specified in an saCkptCheckpointOpen() or saCkptCheckpointOpenAsync() call to obtain the handle checkpointHandle which in turn was used as an input parameter of the saCkptCheckpointSynchronizeAsync() call.

The result of the function is returned in the error parameter.

This function fails with the SA_AIS_ERR_ACCESS return value if the checkpoint identified by `checkpointHandle` in the corresponding invocation of the `saCkptCheckpointSynchronizeAsync()` function was not opened for write mode.

### Return Values

None

### See Also

`saCkptCheckpointSynchronizeAsync()`, `saCkptCheckpointOpen()`, `saCkptCheckpointOpenAsync()`, `saCkptDispatch()`

# 4 Checkpoint Service UML Information Model

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

The Checkpoint Service UML model is implemented by the SA Forum IMM Service ([2]). For further details on this implementation, refer to the SA Forum Overview document ([1]).

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

## 4.1 DN Format for Checkpoint Service UML Classes

**Table 2 DN Formats for Objects of Checkpoint Service Classes**

| Object Class | DN Formats for Objects of the Class |
|---|---|
| `SaCkptCheckpoint` | `"safCkpt=...,*"` |
| `SaCkptReplica` | `"safReplica=...,safCkpt=...,*"` |

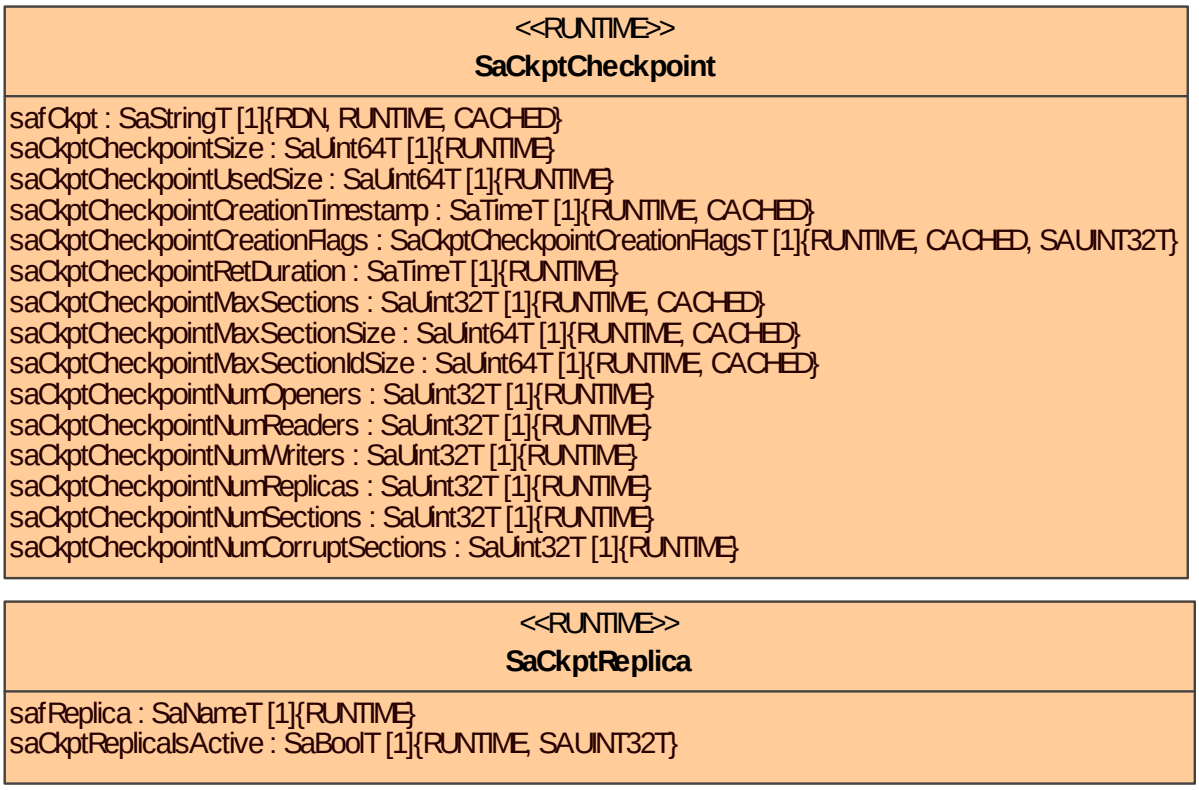
## 4.2 Checkpoint Service UML Classes

The two object classes of the Checkpoint Service UML model are:

- `SaCkptCheckpoint` —This is a runtime object class, which is used to represent a checkpoint.
- `SaCkptReplica` —This is a runtime object class, which is used to represent a checkpoint replica.

FIGURE 1 shows these two 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 (see [1]).

**FIGURE 1** Checkpoint Service UML Classes



# 5 Checkpoint Service Administration API

The Checkpoint Service has no administration interface at the time of publication of this specification.

# 6 Checkpoint Service Alarms and Notifications

The Checkpoint Service produces certain alarms and notifications to convey important information regarding

- its operational and functional state and
- 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 that signify important state or object changes. A management entity should regard notifications, but they do not necessarily require an operator intervention.

The recommended vehicle to be used for producing alarms and notifications is the Notification Service of the Service Availability™ Forum (abbreviated as NTF, see [2]), and hence the various notifications are partitioned into categories as described in this service.

In some cases, this specification uses the word “Unspecified” for values of attributes. This means that the SA Forum has no specific recommendation on the setting, and the vendor may set these attributes to whatever makes sense in the vendor’s context. Such values are generally optional from the CCITT Recommendation X.733 perspective (see [6]).

## 6.1 Setting Common Attributes

The tables presented in Section 6.2 refer to attributes defined in [2]. The following list provides recommendations regarding how to populate these attributes.

- Correlation ids - They are supplied to correlate two notifications that have been generated because of a related cause. This attribute is optional. But in case of alarms that are generated to clear certain conditions, that is, produced with a perceived severity of `SA_NTF_SEVERITY_CLEARED`, the correlation id shall be populated by the application with the notification id that was generated by the Notification Service when the `saNtfNotificationSend()` API was invoked to issued the actual alarm.
- Event time - The application might pass a timestamp or optionally pass an `SA_TIME_UNKNOWN` value in which case the timestamp is provided by the Notification Service.

- NCI id - The notification class identifier is an attribute of type `SaNtfClassIdT`. The `vendorId` portion of the `SaNtfClassIdT` data structure must be set to `SA_NTF_VENDOR_ID_SAF` always. The `majorId` and `minorId` will vary based on the specific SA Forum service and the particular notification. Every SA Forum service shall have a `majorId` as described in the enumeration `SaNtfSafServicesT` of the Notification Service specification.
- Notification id - This attribute is obtained from the Notification Service when a notification is generated, and hence need not be populated by an application.
- Notifying object - DN of the entity generating the notification. This name must conform to the SA Forum AIS naming convention and 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. For details on the SA Forum AIS naming convention, refer to the SA Forum Overview document ([1]).

## 6.2 Checkpoint Service Notifications

The following describes a set of notifications that a Checkpoint Service implementation shall produce.

The notifying object must be set to the DN "`safApp=safCkptService`" for the Checkpoint Service.

The value of the `majorId` field in the notification class identifier (`SaNtfClassIdT`) must be set to `SA_SVC_CKPT` (as defined in the `SaServicesT` enum in [1]) in all notifications generated by the Lock Service.

The `minorId` field within the notification class identifier (`SaNtfClassIdT`) is set distinctly for each individual notification. This field is range-bound, and the used ranges are:

- Alarms: (0x01–0x64)
- State change notifications: (0x65–0xC8)
- Object change notifications: (0xC9–0x12C)
- Attribute change notifications: (0x12D–0x190)

### 6.2.1 Checkpoint Service Alarms

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

### 6.2.2 Checkpoint State Change Notifications

#### *6.2.2.1 Checkpoint Section Resources Exhausted*

**Description**

The maximum number of sections is reached, or no memory is left for write access. In future, configuration parameters may be defined as a part of the Checkpoint Service configuration that will control the generation of this notification.

**Table 3 Checkpoint Section Resources Exhausted Notification**

| NTF Attribute Name | Attribute Type (NTF-Recommended Value) | SA Forum-Recommended Value |
|---|---|---|
| Event Type | Mandatory | `SA_NTF_OBJECT_STATE_CHANGE` |
| Notification Object | Mandatory | LDAP DN of the checkpoint that is full and not available for write access. |
| Notification Class Identifier | NTF-internal | `minorId` = 0x65 |
| Additional Text | Optional | Unspecified |
| Additional Information | Optional | Unspecified |
| Source Indicator | Mandatory | `SA_NTF_OBJECT_OPERATION` or `SA_NTF_UNKNOWN_OPERATION` |
| Changed State Attribute ID | Optional | `SA_CKPT_CHECKPOINT_STATUS` |
| Old Attribute Value | Optional | Unspecified |
| New Attribute Value | Mandatory | `SA_CKPT_SECTION_RESOURCES_EXHAUSTED` |

#### *6.2.2.2 Checkpoint Section Resources Available*

### Description

Memory is available for write access, or at least one section within the checkpoint is available, after having recovered from a preceding SA_CKPT_SECTION_RESOURCES_EXHAUSTED condition. This notification is generated only to indicate a recovery from an SA_CKPT_SECTION_RESOURCES_EXHAUSTED condition and not otherwise. In future, configuration parameters may be defined as a part of the Checkpoint Service configuration that will control the generation of this notification.

**Table 4 Checkpoint Section Resources Available Notification**

| NTF Attribute Name | Attribute Type (NTF-Recommended Value) | SA Forum-Recommended Value |
|---|---|---|
| Event Type | Mandatory | SA_NTF_OBJECT_STATE_CHANGE |
| Notification Object | Mandatory | LDAP DN of the checkpoint which is available for write access. |
| Notification Class Identifier | NTF internal | minorId = 0x66 |
| Additional Text | Optional | Unspecified |
| Additional Information | Optional | Unspecified |
| Source Indicator | Mandatory | SA_NTF_OBJECT_OPERATION or SA_NTF_UNKNOWN_OPERATION |
| Changed State Attribute ID | Optional | SA_CKPT_CHECKPOINT_STATUS |
| Old Attribute Value | Optional | SA_CKPT_SECTION_RESOURCES_EXHAUSTED |
| New Attribute Value | Mandatory | SA_CKPT_SECTION_RESOURCES_AVAILABLE |

# 7 Checkpoint Service Management Interface

Currently, an SNMP MIB interface is defined for the Checkpoint Service. Other management access methods to the Checkpoint Service may be added in future versions of this specification.

## 7.1 Checkpoint Service MIB (SAF-CKPT-SVC-MIB)

The Checkpoint Service MIB contains two read-only tables, `saCkptCheckpointTable` and `saCkptNodeReplicaLocTable`.

The `saCkptCheckpointTable` contains runtime attributes for the currently created checkpoints in the cluster, which include all checkpoints in the cluster that have not been unlinked as well as the ones that have been unlinked but are still in-use within the cluster.
This table mimics the UML runtime object class `saCkptCheckpoint`, as described in Section 4.2, in terms of the objects contained in the table.

The `saCkptNodeReplicaLocTable` shows on which nodes a particular checkpoint has replicas.
This table mimics the UML runtime object class `SaCkptReplica`, as described in Section 4.2, in terms of the objects contained in the table.

Additionally, the Checkpoint Service MIB also defines SNMP traps that correspond to the various notifications for the service, which are described in Chapter 6 of this specification.

For a detailed description of the various objects of this MIB, refer to the SAF-CKPT-SVC-MIB that can be downloaded from http://www.saforum.org/specification/download/get_spec.

# Index of Definitions