Service Availability™ Forum Application Interface Specification

Volume 4: Checkpoint Service SAI-AIS-CKPT-B.01.01

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

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

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

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

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

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

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

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

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

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

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

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

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

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

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

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

5. OWNERSHIP OF SPECIFICATION AND COPYRIGHTS. The Specification and all worldwide copyrights therein are the exclusive property of Licensor. You may not remove, obscure, or alter any copyright or other proprietary rights notices that are in or on the copy of the Specification you download. You must reproduce all such notices on all copies of the Specification you make. Licensor may make changes to the Specification, or to items referenced therein, at any time without notice. Licensor is not obligated to support or update the Specification.

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

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

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

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

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

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

Table of Contents Volume 4, 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. It is intended for use by implementors 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) and with the Service Availability™ Forum System Management Specification, which is still under development.

1.2 AIS Documents Organization

The Application Interface Specification is organized into the following volumes:

Volume 1, the Overview document, provides a brief guide to the remainder of the Application Interface Specification. It describes the objectives of the AIS specification as well as programming models and definitions that are common to all specifications. Additionally, it contains an overview of the Availability Management Framework and of the other services as well as the system description and conceptual models, including the physical and logical entities that make up the system. Volume 1 also contains a chapter that describes the main abbreviations, concepts and terms used in the AIS documents.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisOverview.B0101.pdf

Volume 2 describes the Availability Management Framework API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisAmf.B0101.pdf

Volume 3 describes the Cluster Membership Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisClm.B0101.pdf

Volume 4 (this volume) describes the Checkpoint Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisCkpt.B0101.pdf

Volume 5 describes the Event Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisEvt.B0101.pdf

Volume 6 describes the Message Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisMsg.B0101.pdf

Volume 7 describes the Lock Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisLck.B0101.pdf

1.3 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.

1.4 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.

1.5 Additional Information

1.5.1 Member Companies

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

1.5.2 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, which is typically stored in main memory rather than on disk for performance reasons. A given checkpoint may have several checkpoint replicas stored on different nodes 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.

3 SA Checkpoint Service API

3.1 Checkpoint Service Model

3.1.1 Checkpoints

The Checkpoint Service manages a set of entities, called **checkpoints**, that processes use to save their state. A given 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 using the *saCkptCheckpointOpen()* or the *saCkptCheckpointOpenAsync()* API functions.

A process can delete checkpoints using the *saCkptCheckpointUnlink()* function. After a checkpoint has been deleted, it cannot be accessed using its global name, but processes that have the checkpoint opened can continue to access it until they close it (*saCkptCheckpointClose()*). This means that, after the checkpoint is deleted, resources associated with the checkpoint are freed only when the last process that has opened the checkpoint closes it.

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**. The maximum number of sections 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, 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.

Within a checkpoint, sections of different sizes may coexist but a maximum section size is specified for a checkpoint when it is created. The size of a section can be changed dynamically as the result of the invocation of the *saCkptCheckpointWrite()* or *saCkptSectionOverwrite()* functions. Sections contain raw data that is not encoded by the Checkpoint Service. It is the responsibility of the process to encode the section contents if heterogeneity is a concern.

Sections have an **expiration time** that 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**. At most one checkpoint replica for a particular checkpoint may reside on a given cluster node. A given checkpoint may have several checkpoint replicas (or copies) that reside on different cluster nodes.

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 given checkpoint can have negative effects on the performance of checkpoint write operations. To give flexibility to implementors 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, 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 given checkpoint and then writes D2 in another section of the same checkpoint. Later on, if the same process or another process running on another cluster node reads both sections entirely, and it sees D2, then 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 there are no partial updates in a replica: a replica is either updated with all data specified in the write call or is not updated at all.
Hence, the Checkpoint Service guarantees that all replicas of a checkpoint created with the synchronous update option are identical.

It is not specified from which replica checkpoint data is read from.

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 3.3.2.1 on page 18.

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, there is at most one active replica.

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.

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 ensure that the Checkpoint Service **synchronizes** all checkpoint replicas by using an invocation of the *saCkptCheckpointSynchronize()* function to propagate checkpoint data to all of the checkpoint replicas.

There are 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 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 to Section 3.3.2.1 on page 18.

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 *saCkptCheckpointDelete()*. 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 3.3.2.1 on page 18.

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.

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, optimal performance can be achieved only if the application informs the Checkpoint Service about which replica should be active at a particular time. This can be done for checkpoints having the collocated attribute. Such a checkpoint is named a **collocated checkpoint.** When a collocated checkpoint is created, there is no active replica 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, there is no active replica 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.

If there is no active replica, each of the operations write, overwrite, creation or deletion of a section, and read will return an error.

The management of replicas of collocated checkpoints and whether they are active or not is mainly the duty of applications. The replicas of a collocated checkpoint are only created by the applications via open calls, provided that no local replica already exists. The Checkpoint Service does not create replicas other than the ones explicitly created by the applications via the 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 they are 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. The Checkpoint Service may create replicas other than the ones that may be created when opening a checkpoint. This can be useful to enhance the availability of checkpoints. For example, if there are at a certain point in time two replicas, 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.

If there is no active replica for a checkpoint with the asynchronous update option, each of the operations write, overwrite, creation or deletion of a section, and read will return an error.

3.1.8 Persistence of Checkpoints

As has been stated in Section 2.1 on page 11, the Checkpoint Service stores checkpoint data in the main memory of the cluster 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. This can be caused by administrative actions or node failures.

3.2 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

3.3 Type Definitions

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

3.3.1 Handles

3.3.1.1 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.

3.3.1.2 SaCkptCheckpointHandleT

typedef SaUint64T SaCkptCheckpointHandleT;

The type of the handle of a checkpoint.

3.3.1.3 SaCkptSectionIterationHandleT

typedef SaUint64T SaCkptSectionIterationHandleT;

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

3.3.2 Checkpoint Types

3.3.2.1 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 the replica.

- SA_CKPT_WR_ACTIVE_REPLICA - The specification of this flag at creation time creates a checkpoint with the asynchronous update option and providing atomicity when updating replicas: 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, that 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 asynchronous and the partial update option. Any of the operations that modify the checkpoint will be performed on the active checkpoint replica before it returns. However, there is no guarantee of atomicity per checkpoint replica, that is, if the operation of writing does not complete, 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 16.

The flags SA_CKPT_WR_ALL_REPLICAS, SA_CKPT_WR_ACTIVE_REPLICA, and SA_CKPT_WR_ACTIVE_REPLICA_WEAK are mutual exclusive. A value or parameter of the type *SaCkptCheckpointCreationFlagsT* is either one of the mutual exclusive flags or the bitwise OR of one of these mutual exclusive flags and the SA_CKPT_CHECKPOINT_COLLOCATED flag.

3.3.2.2 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 follow:

- *creationFlags* - These flags specify the collocation attribute of checkpoint replicas and the manner of both synchronizing write operations on the checkpoint replicas using the *saCkptCheckpointWrite()* and *saCkptSectionOverwrite()* functions and creating and deleting sections using the *saCkptSectionCreate()*

and *saCkptSectionDelete()* functions. For details, refer to Section 3.3.2.1 on page 18.

- *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* - Maximum number of sections in the checkpoint. Every checkpoint has at least one section. If and only if *maxSections* is 1, then a default section is automatically created and is identified by the special identifier SA_CKPT_DEFAULT_SECTION_ID.
- *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.

Note that *checkpointSize* =< *maxSections* * *maxSectionSize*.

3.3.2.3 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.
- SA_CKPT_CHECKPOINT_CREATE - The checkpoint is created if it does not already exist.

3.3.3 Section Types

3.3.3.1 SaCkptSectionIdT

#define SA_CKPT_DEFAULT_SECTION_ID {0, NULL}

#define SA_CKPT_GENERATED_SECTION_ID {0, NULL}

These special constants define the identifier of the default section and the identifier of a generated section, as defined by the *SaCkptSectionIdT* structure below and

returned by an invocation of the *saCkptSectionCreate()* function. The actual identifier of the generated section is present in the *sectionId* field of the *SaCkptSectionCreationAttributesT,* when the invocation of the *saCkptSectionCreate()* function returns.

```
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.

3.3.3.2 SaCkptSectionCreationAttributesT

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

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

- *sectionId* - [in/out] A structure of the type *saCkptSectionIdT* that identifies the section that is to be created. If it contains the special value SA_CKPT_GENERATED_SECTION_ID, the Checkpoint Service automatically generates a new identifier and changes the values of the fields in the structure *sectionId*.
- *expirationTime* - [in] The absolute time after which the Checkpoint Service deletes the section automatically. The *expirationTime* can be specified when a section is created as well as set or modified later via the *saCkptSectionExpirationTimeSet()* call. If *expirationTime* has the special value SA_TIME_END, the Checkpoint Service never deletes the section automatically.

3.3.3.3 SaCkptSectionStateT

typedef enum {

SA_CKPT_SECTION_VALID = 1,

SA_CKPT_SECTION_CORRUPTED = 2

} SaCkptSectionStateT;

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

3.3.3.4 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 identifier of the section.
- *expirationTime* - The absolute time at which the section will be deleted.
- *sectionSize* - The size of the section.
- *sectionState* - The state of the section. A section is either in the SA_CKPT_SECTION_VALID state or 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.

3.3.3.5 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 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.

3.3.4 IoVector Types

3.3.4.1 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 identifier of the section to be written to or read from.
- *dataBuffer* - A pointer to a buffer containing the data to be written or read.

- *dataSize* - Size of the data in bytes to be written to, or read from, the buffer *dataBuffer*. The 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 that is to be written or read.
- *readSize* - Used by *saCkptCheckpointRead()* to record the number of bytes of data that have been read; otherwise, this field is not used.

3.3.5 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.

3.3.6 SaCkptCallbacksT

The *SaCkptCallbacksT* structure is defined as follows:

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

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

3.4 Library Life Cycle

3.4.1 saCkptInitialize()

Prototype

SaAisErrorT saCkptInitialize(

*SaCkptHandleT *ckptHandle,*

*const SaCkptCallbacksT *ckptCallbacks,*

*SaVersionT *version*

);

Parameters

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

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

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

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 *ckptHandle* is returned 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 required *releaseCode,* and a major version >= the required *majorVersion*, SA_AIS_OK is returned. In this case, the *version* parameter is set by this function to:

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

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

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

if (implementation supports the required *releaseCode*)

releaseCode = required *releaseCode*

else *{*

if (implementation supports *releaseCode* higher than the required *releaseCode*)

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

else

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

}

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

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

Return Values

SA_AIS_OK - The function completed successfully.

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

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

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* parameter is not compatible with the version of the Checkpoint Service implementation.

See Also

saCkptSelectionObjectGet(), saCkptDispatch(), saCkptFinalize()

3.4.2 saCkptSelectionObjectGet()

Prototype

SaAisErrorT saCkptSelectionObjectGet(

SaCkptHandleT ckptHandle,

*SaSelectionObjectT *selectionObject*

);

Parameters

ckptHandle - [in] The handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

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

Description

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

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

The *selectionObject* returned by *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 didn't.

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).

See Also

saCkptInitialize(), saCkptDispatch(), saCkptFinalize()

3.4.3 saCkptDispatch()

Prototype

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

Parameters

ckptHandle - [in] The handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

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, as defined in volume 1 of the AIS specification.

Description

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

Return Values

SA_AIS_OK - The function completed successfully.

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

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

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.

See Also

saCkptInitialize(), saCkptSelectionObjectGet()

3.4.4 saCkptFinalize()

Prototype

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

Parameters

ckptHandle - [in] The handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

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 returns successfully, the *saCkptFinalize()* function 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 callbacks related to the particular handle.

After *saCkptFinalize()* is called, the selection object is no longer valid. Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

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 didn't.

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()

3.5 Checkpoint Management

3.5.1 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, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

invocation - [in] A parameter designates a particular invocation of the response callback.

checkpointName - [in] A pointer to the name of the checkpoint that identifies a checkpoint globally in a cluster.

checkpointCreationAttributes - [in] A pointer to the creation attributes of a checkpoint.

If the intent is 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 intent is to open and create a checkpoint if it does not exist, *checkpointCreationAttributes* must contain the attributes for the checkpoint, and the SA_CKPT_CHECKPOINT_CREATE flag in *checkpointOpenFlags* must be set. If the checkpoint already exists, the creation attributes must match the ones used at creation time.

checkpointOpenFlags - [in] The value of this parameter is constructed by a bitwise OR of the flags defined by the *SaCkptCheckpointOpenFlagsT* type in Section 3.3.2.3 on page 20.

timeout - [in] The *saCkptCheckpointOpen()* invocation is considered to have failed if it does not complete by the time specified. A checkpoint replica may still be created.

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 in *checkpointHandle* 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.

Description

The *saCkptCheckpointOpen()* and *saCkptCheckpointOpenAsync()* 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 created first.

An invocation of *saCkptCheckpointOpen()* is blocking. A new checkpoint handle is returned upon completion. A checkpoint can be opened multiple times for reading and 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 synchronization flag SA_CKPT_WR_ALL_REPLICAS, then the checkpoint replica must be identical to the other checkpoint replicas.
- Otherwise, the data in the checkpoint replica is synchronized using the data in the active checkpoint replica.

When a checkpoint is opened using the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* function, some combination of the creation flags, defined in *SaCkptCheckpointCreationFlagsT,* are bitwise ORed together to provide the value of the *creationFlags* field of the *checkpointCreationAttributes* parameter.

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 s*aCkptCheckpointOpenAsync()* 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 didn't.

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 initialization with *saCkptInitialize()* was incomplete, since the *saCkptCheckpointOpenCallback()* callback function is missing. This return value only applies to the s*aCkptCheckpointOpenAsync()* 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.

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, and the checkpoint, designated by *checkpointName,* does not exist.

SA_AIS_ERR_EXIST - The checkpoint already exists and the *checkpointCreationAttributes* creation attributes are different from the ones used at creation time.

SA_AIS_ERR_BAD_FLAGS - The *checkpointOpenFlags* parameter is invalid.

See Also

SaCkptCheckpointOpenCallbackT, saCkptCheckpointClose(), saCkptInitialize()

3.5.2 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.

checkpointHandle - [in] The handle that designates the checkpoint.

error - [in] This parameter indicates whether the *saCkptCheckpointOpenAsync()* function was successful. 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 didn't.

- SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may try again.
- 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, and the checkpoint, designated by *checkpointName,* does not exist.
- SA_AIS_ERR_EXIST - The checkpoint already exists, and the *checkpointCreationAttributes* creation attributes are different from the ones used at creation time.
- SA_AIS_ERR_BAD_FLAGS - The *checkpointOpenFlags* parameter is invalid.

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 issuing an *saCkptDispatch()* call on the handle *ckptHandle,* which 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()

3.5.3 saCkptCheckpointClose()

Prototype

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

Parameters

checkpointHandle - [in] The handle that designates the checkpoint to close. The handle *checkpointHandle* must have been obtained previously by the invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* functions.

Description

This API function closes the checkpoint, designated by *checkpointHandle,* which was opened by an earlier invocation of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* function.

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

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 a *saCkptCheckpointUnlink()* function, or
- the checkpoint will be deleted when the retention duration expires if no process opens it in the meantime.

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

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

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.

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 didn't.

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 via 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.

See Also

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

3.5.4 saCkptCheckpointUnlink()

Prototype

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

Parameters

ckptHandle - [in] The handle, obtained through the *saCkptInitialize()* function, designating this particular initialization of the Checkpoint Service.

checkpointName - [in] A pointer to the name of the checkpoint that is to be unlinked.

Description

This function deletes an existing checkpoint, identified by *checkpointName,* from the cluster.

After completion of the invocation,

- The name *checkpointName* is no longer valid, that is, any invocation of a function of the Checkpoint Service API that uses the checkpoint name returns an error, unless a checkpoint is re-created with this name. The checkpoint is re-created by specifying the same name of the checkpoint to be unlinked in an open call with the SA_CKPT_CHECKPOINT_CREATE flag set. 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 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 didn't.

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 *checkpointName,* does not exist.

See Also

saCkptCheckpointClose()

3.5.5 saCkptCheckpointRetentionDurationSet()

Prototype

SaAisErrorT saCkptCheckpointRetentionDurationSet(

SaCkptCheckpointHandleT checkpointHandle,

SaTimeT retentionDuration

);

Parameters

checkpointHandle - [in] The checkpoint whose retention time is being set. The handle *checkpointHandle* must have been obtained previously by the invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* functions.

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

Description

The function *saCkptCheckpointRetentionDurationSet()* sets the retention duration of the checkpoint, designated by *checkpointHandle,* to *retentionDuration*. When no more processes have the checkpoint open, and if the checkpoint is not opened by any process within the retention duration, the Checkpoint Service automatically deletes 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 didn’t.

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 via 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_BAD_OPERATION - The retention duration of the checkpoint, designated by *checkpointHandle,* cannot be changed as the checkpoint has been unlinked.

See Also

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

3.5.6 saCkptActiveReplicaSet()

Prototype

SaAisErrorT saCkptActiveReplicaSet(

SaCkptCheckpointHandleT checkpointHandle

);

Parameters

checkpointHandle - [in] The handle that designates a checkpoint. The handle *checkpointHandle* must have been obtained previously by the invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* functions.

Description

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

The local checkpoint replica will become the active replica after an invocation of this function.

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

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 didn't.

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 via 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, designated by *checkpointHandle,* was not opened for write mode.

SA_AIS_ERR_BAD_OPERATION - The checkpoint, designated by *checkpointHandle,* was not created as a collocated checkpoint with the asynchronous update option.

See Also

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

3.5.7 saCkptCheckpointStatusGet()

Prototype

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

Parameters

checkpointHandle - [in] The handle of the checkpoint whose status is to be returned. The handle *checkpointHandle* must have been obtained previously by the invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* functions.

checkpointStatus -[out] A pointer to a *SaCkptCheckpointDescriptorT* structure, defined in Section 3.3.5 on page 24, in the address space of the invoking process, that contains the checkpoint status information that is to be returned.

Description

This function retrieves the *checkpointStatus* of the checkpoint designated by *checkpointHandle*.

If the checkpoint was created using either SA_CKPT_WR_ACTIVE_REPLICA or SA_CKPT_WR_ACTIVE_REPLICA_WEAK option, the checkpoint status is obtained from the active replica. If the checkpoint was created using the SA_CKPT_WR_ALL_REPLICAS option, the Checkpoint Service determines the replica from which to obtain the checkpoint status.

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 didn’t.

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 via 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.

See Also

saCkptCheckpointOpen(), saCkptCheckpointOpenAsync(), SaCkptCheckpointOpenCallbackT

3.6 Section Management

3.6.1 saCkptSectionCreate()

Prototype

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

Parameters

checkpointHandle - [in] The handle of the checkpoint that is to hold the section to be created. The handle *checkpointHandle* must have been obtained by a previous invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* function with the SA_CKPT_CHECKPOINT_WRITE flag set.

sectionCreationAttributes - [in] A pointer to a *SaCkptSectionCreationAttributesT* structure, as defined in Section 3.3.3.2 on page 21, that contains the in/out field *sectionId* and the in field *expirationTime*.

initialData - [in] 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. Initial size can be at most *maxSectionSize,* as specified by the checkpoint creation attributes in *saCkptCheckpointOpen()*.

Description

This function creates a new section in the checkpoint referred to by *checkpointHandle* as long as the total number of existing sections is less than the maximum number of sections specified by the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* API call. Unlike a checkpoint, a section does not need to be opened for access. The section will be deleted by the Checkpoint Service 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 identifier 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.

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 didn’t.

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 via 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, designated by *checkpointHandle,* was not opened for write mode.

SA_AIS_ERR_EXIST - The section, defined in *sectionCreationAttributes,* already exists, or the checkpoint was created to have only one section.

See Also

saCkptSectionDelete(), saCkptCheckpointOpen(), saCkptCheckpointOpenAsync(), SaCkptCheckpointOpenCallbackT

3.6.2 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 one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* functions with the SA_CKPT_CHECKPOINT_WRITE flag set.

sectionId - [in] A pointer to the identifier of the section that is to be deleted.

Description

This function deletes a section in the checkpoint referred to 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 SA_CKPT_DEFAULT_SECTION_ID, cannot be deleted by invoking the *saCkptSectionDelete()* 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 didn't.

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 via 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 - There is no active replica, or the section, identified by *sectionId,* does not exist.

SA_AIS_ERR_ACCESS -The checkpoint, designated by *checkpointHandle,* was not opened for write mode.

See Also

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

3.6.3 saCkptSectionExpirationTimeSet()

Prototype

SaAisErrorT saCkptSectionExpirationTimeSet(

SaCkptCheckpointHandleT checkpointHandle,

const SaCkptSectionIdT sectionId,*

SaTimeT expirationTime

);

Parameters

checkpointHandle - [in] The handle of the checkpoint containing the section for which the expiration time is to be set. The handle *checkpointHandle* must have been obtained previously by the invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* functions.

sectionId - [in] A pointer to the identifier of the section for which the expiration time is to be set.

expirationTime - [in] The expiration time that is to be set for the section, designated by *sectionId*. 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.

Description

This function sets the expiration time of the section, identified by *sectionId,* within the checkpoint with 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.

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 didn't.

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 via 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_ACCESS -The checkpoint, designated by *checkpointHandle,* was not opened for write mode.

SA_AIS_ERR_NOT_EXIST - There is no active replica, or the section, identified by *sectionId,* does not exist.

See Also

saCkptSectionCreate(), saCkptCheckpointOpen(), *SaCkptCheckpointOpenCallbackT*

3.6.4 saCkptSectionIterationInitialize()

Prototype

SaAisErrorT saCkptSectionIterationInitialize(

SaCkptCheckpointHandleT checkpointHandle,

SaCkptSectionsChosenT sectionsChosen,

SaTimeT expirationTime,

*SaCkptSectionIterationHandleT *sectionIterationHandle*

);

Parameters

checkpointHandle - [in] The checkpoint handle, which must have been obtained previously by an invocation of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* functions.

sectionsChosen - [in] A predicate, defined by the *SaCkptSectionsChosenT* structure in Section 3.3.3.5 on page 23, that describes the sections that are to be chosen during an iteration.

expirationTime - [in] An absolute time used by *sectionsChosen,* as described above. This field is not used when *sectionsChosen* is SA_CKPT_SECTIONS_FOREVER, SA_CKPT_SECTIONS_CORRUPTED or SA_CKPT_SECTIONS_ANY.

sectionIterationHandle - [out] A pointer to the section iteration handle, allocated in the address space of the invoking process. If this function returns successfully, the Checkpoint Service stores in *sectionIterationHandle* the handle that the process uses in subsequent invocations of the *saCkptSectionIterationNext()* and *saCkptSectionIterationFinalize()* functions for stepping through the sections in the checkpoint designated by *checkpointHandle*.

Description

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

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 didn't.

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 via 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.

See Also

saCkptSectionIterationNext(), saCkptSectionIterationFinalize(), saCkptCheckpointOpen(), *saCkptCheckpointOpenAsync()*

3.6.5 saCkptSectionIterationNext()

Prototype

SaAisErrorT saCkptSectionIterationNext(

SaCkptSectionIterationHandleT sectionIterationHandle,

*SaCkptSectionDescriptorT *sectionDescriptor*

);

Parameters

sectionIterationHandle - [in] The section iteration handle obtained via an invocation of the *saCkptSectionIterationInitialize()* function for stepping through the sections in the checkpoint.

sectionDescriptor - [out] A pointer to a *SaCkptSectionDescriptorT* structure, defined in Section 3.3.3.4 on page 22, that is allocated by the Checkpoint Service in the address space of the invoking process and that contains information about the section.
It is up to the Checkpoint Service to release the memory for *sectionDescriptor* and the section identifier (*sectionDescriptor->sectionId->id*). Releasing this memory is usually done at the next invocation of the *saCkptSectionIterationNext()* function. The Checkpoint Service also releases this memory when the *saCkptSectionIterationFinalize()* function is invoked, the corresponding checkpoint is closed, or the handle of this particular initialization of the Checkpoint Service is finalized.

Description

This function iterates over an internal table of sections using the handle *sectionIterationHandle,* which was obtained via the *saCkptSectionIterationInitialize()* function. When the function returns, *sectionDescriptor* 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 made: 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 didn't.

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 was not obtained via the *saCkptSectionIterationInitialize()* function, or *saCkptSectionIterationFinalize()* has already been invoked.
- The checkpoint, identified by *checkpointHandle,* that 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_NOT_EXIST - No active replica exists.

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

See Also

saCkptSectionIterationInitialize(), saCkptSectionIterationFinalize(), saCkptCheckpointOpen(), *saCkptCheckpointOpenAsync()*

3.6.6 saCkptSectionIterationFinalize()

Prototype

SaAisErrorT saCkptSectionIterationFinalize(

SaCkptSectionIterationHandleT sectionIterationHandle

);

Parameters

sectionIterationHandle - [in] The section iteration handle obtained via an invocation of the *saCkptSectionIterationInitialize()* function and identifying the iteration to be finalized.

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 didn't.

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 was not obtained via the *saCkptSectionIterationInitialize()* function, or *saCkptSectionIterationFinalize()* has already been invoked.
- The checkpoint, identified by *checkpointHandle,* that 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.

See Also

saCkptSectionIterationInitialize(), saCkptSectionIterationNext(), saCkptCheckpointOpen(), *saCkptCheckpointOpenAsync()*

3.7 Data Access

3.7.1 saCkptCheckpointWrite()

Prototype

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

Parameters

checkpointHandle - [in] The handle to the checkpoint that is to be written to. The handle *checkpointHandle* must have been obtained by a previous invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* function with the SA_CKPT_CHECKPOINT_WRITE flag set.

ioVector - [in] A pointer to a vector with elements *ioVector*[0], ..., *ioVector*[*numberOfElements* - 1]. Each element is of the type *SaCkptIOVectorElementT,* defined in Section 3.3.4.1 on page 23, which contains the following fields: *sectionId, dataBuffer, dataSize*, *dataOffset* and *readSize.* If *sectionId* is equal to SA_CKPT_DEFAULT_SECTION_ID, then 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 not used by the *saCkptCheckpointWrite()* function.

numberOfElements - [in] Size of the *ioVector*.

erroneousVectorIndex - [out] A pointer to an index, stored in the caller's address space, of the first *iovector* element that makes the invocation fail. If the index is set to NULL or if the invocation succeeds, the field remains unchanged.

Description

This function writes data from the memory regions specified by *ioVector* into a checkpoint:

- If this checkpoint has been created with the SA_CKPT_WR_ALL_REPLICAS property, all of the checkpoint replicas have been updated when the invocation returns. If the invocation does not complete or returns with an error, 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. Other checkpoint replicas are updated asynchronously. If the invocation does not complete or returns with an error, 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. 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, 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 *ioVector*s are written in order from *ioVector*[0] to *ioVector*[*numberOfElements* - 1]. As a result of this invocation, some sections might grow.

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 didn't.

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 via 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 - There is no active replica, or a section, identified by *sectionId* in *ioVector,* does not exist.

SA_AIS_ERR_ACCESS -The checkpoint, designated by *checkpointHandle,* was not opened for write mode.

See Also

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

3.7.2 saCkptSectionOverwrite()

Prototype

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

Parameters

checkpointHandle - [in] The handle that designates the checkpoint that is written to. The handle *checkpointHandle* must have been obtained by a previous invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* functions with the SA_CKPT_CHECKPOINT_WRITE flag set.

sectionId - [in] A pointer to an identifier for the section that is to be overwritten. If this pointer points to SA_CKPT_DEFAULT_SECTION_ID, the default section is updated.

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, which becomes the new size for this section.

Description

This function is similar to *saCkptCheckpointWrite()* except that it overwrites only a single section. As a result of this invocation, the previous data and size of the section will change. This function may be invoked even if there was no prior invocation of *saCkptCheckpointWrite()*.

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 didn't.

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 via 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 - There is no active replica, or the section, identified by *sectionId,* does not exist.

SA_AIS_ERR_ACCESS -The checkpoint, designated by *checkpointHandle,* was not opened for write mode.

See Also

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

3.7.3 saCkptCheckpointRead()

Prototype

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

Parameters

checkpointHandle - [in] The handle to the checkpoint that is to be read. The handle *checkpointHandle* must have been obtained by a previous invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenCallback()* function.

ioVector - [in/out] A pointer to a vector that contains elements *ioVector*[0], ..., *ioVector*[*numberOfElements* - 1]. Each element is of the type *saCkptIOVectorElementT,* defined in Section 3.3.4.1 on page 23, and containing the following fields:

- *sectionId* - [in] The identifier of the section to be read from.
- *dataBuffer* - [in/out] A pointer to a buffer containing the data to be read to. 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 deallocated by the invoker.
- *dataSize* - [in] Size of the data to be read to the buffer designated by *dataBuffer*. 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 that is to be read.

- *readSize* - [out] Used by *saCkptCheckpointRead()* to record the number of bytes of data that have been read; otherwise, this field is not used.

numberOfElements - [in] The size of the *ioVector*.

erroneousVectorIndex - [out] A pointer to an index, in the caller's address space, of the first vector element that causes the invocation to fail. If the invocation succeeds, then *erroneousVectorIndex* is NULL and should be ignored.

Description

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

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

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 didn't.

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 via 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 - There is no active replica, or a section, identified by *sectionId* in *ioVector,* does not exist.

SA_AIS_ERR_ACCESS -The checkpoint, designated by *checkpointHandle,* was not opened for read mode.

See Also

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

3.7.4 saCkptCheckpointSynchronize(), saCkptCheckpointSynchronizeAsync()

Prototype

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

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

Parameters

checkpointHandle -[in] The handle of the checkpoint that is to be synchronized. The handle *checkpointHandle* must have been obtained by a previous invocation of one of the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* functions with the SA_CKPT_CHECKPOINT_WRITE flag set.

invocation - [in] This parameter designates a particular invocation of the response callback.

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.

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()*.

There is no guarantee that new operations, applied while the synchronization is in progress, will be propagated when the synchronization operation completes. In the case where new operations are issued concurrently to these calls, there is no guarantee that the replicas are all identical when *saCkptCheckpointSynchronize()* returns, or the *saCkptCheckpointSynchronizeCallback()* callback is invoked.

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

Only those processes that have the checkpoint open in SA_CKPT_CHECKPOINT_WRITE mode may invoke this function.

For the *saCkptCheckpointSynchronize()* function, when the timeout expires, there is no guarantee that the checkpoint replicas have been synchronized.

For the *saCkptCheckpointSynchronizeAsync()* function, completion of the 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.

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 didn't.

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 via 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 initialization with *saCkptInitialize()* was incomplete, since the *saCkptCheckpointSynchronizeCallback()* callback function is missing. This return value only applies to the s*aCkptCheckpointSynchronizeAsync()* 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, designated by *checkpointHandle,* was not opened for write mode.

SA_AIS_ERR_BAD_OPERATION - The checkpoint, designated by *checkpointHandle,* was not created with the asynchronous update option.

See Also

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

3.7.5 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.

error - [in] This parameter indicates whether the *saCkptCheckpointSynchronize()* function was successful. 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 didn't.
- 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_NOT_EXIST - No active replica exists.
- SA_AIS_ERR_ACCESS - The checkpoint, designated by *checkpointHandle* in the corresponding *saCkptCheckpointSynchronizeAsync()* call, was not opened for write mode.
- SA_AIS_ERR_BAD_OPERATION - The checkpoint, designated by *checkpointHandle* in the corresponding *saCkptCheckpointSynchronizeAsync()* call, was not created with the asynchronous update option.

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 issuing an *saCkptDispatch()* call on the handle *ckptHandle,* associated with the *saCkptCheckpointSynchronizeAsync()* call. Associated means here that *ckptHandle* was specified in the *saCkptCheckpointOpen()* or *saCkptCheckpointOpenAsync()* call, leading to the handle *checkpointHandle,* which was used as an input parameter of the *saCkptCheckpointSynchronizeAsync()* call. The result of the function is returned in the error parameter.

Return Values

None.

See Also

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