OpenClovis
Software Development Kit (SDK)
Service Description and API Reference for Checkpoint Service (CPS)

For OpenClovis SDK Release 2.3 V0.4

Document Revision Date: March 20, 2007

Contents

Chapter 1

Functional Overview

The OpenClovis Checkpoint Service (CPS) is a high availability infrastructure component that provides synchronization of run-time data and context that ensures a seamless failover or switchover of applications. CPS allows the application to store its internal state and retrieve the information immediately, at regular intervals, during a failover, a switchover, or at a restart. It provides the facility for processes to record Checkpoint data incrementally to protect an application against failures. While recovering from fail-over, restart, or switch-over situations, the Checkpoint data can be retrieved and execution can be resumed from the state recorded before the failure. CPS maintains at least one backup replica for a Checkpoint (provided resource), so that the information is not lost. A Checkpoint can be replicated immediately on a different blade and stored indefinitely, or it can be consumed immediately. CPS provides two types of Checkpointing:

- File-based/Library-based Checkpointing - The Checkpoint is stored in the persistent memory. Such Checkpoints survive through node restart, application restart or failover conditions.
- Server based Checkpointing - The Checkpoint in stored in the main memory. Such Checkpoints survive through application restart/failover situations only.

Chapter 2

Service Model

2.1 Usage Model

The users of CPS are usually highly available applications. The active application Checkpoints its state (or some data) based on certain application dependent logic and the standby counterpart reads the Checkpointed information, when an application logic dependent logic is activated.

CPS neither interprets the stored information nor provides any logic on when to read or write a Checkpoint (an application that has registered for immediate consumption, is an exception.) This logic is handled the application.

The Checkpointed information is replicated by CPS in persistent memory (in case of file based Checkpointing) or in replica nodes (server-based Checkpointing.)

The users can also choose for a trigger from CPS, by registering a callback function. This callback function is invoked when the Checkpoint that the user is interested in, is updated. This is also referred to as re-registering for immediate consumption.

2.1.1 Use cases

File/Library based Checkpointing

CPS replicates the information in the persistent memory. This implies that the user application can survive application restart and node restart scenarios. Typically, this form of Checkpointing is used when there are no other nodes in the system where the replicas can be stored.

Server based Checkpointing

CPS replicates the information on other nodes in the system where the replicas can be stored (if it is available). Typically, this form of Checkpointing is used by the applications that need to recover from fail-over or switch-over situations. Server based Checkpointing supports three types of Checkpoints:

- Synchronous Checkpoints - CPS ensures that such Checkpoints provides consistency of data across replicas.
- Asynchronous non-collocated Checkpoints - CPS does not ensure consistency of data across replicas but the performance while writing to a Checkpoint (speed) is faster. In asynchronous non-collocated Checkpoints, the CPS decides where the active replica resides.

- Asynchronous collocated Checkpoints - These are similar to asynchronous non-collocated Checkpoints except that the control where the active replica resides, lies with the user application.

2.2 Functional Description

CPS provides a facility to store information that can be read by users immediately, or, later depending on the logic of the user application. The CPS functional description can be divided into the following sections based on the type of service used.

- File/Library-based Checkpointing
- Server-based Checkpointing

2.2.1 File/Library based Checkpointing

Checkpoint Life cycle

For using the services of CPS, the application has to initialize the CPS using the `clCkptLibraryInitialize()` function. `ClCkptLibraryInitialize()` returns a handle, `ckptSvcHandle`, that can be used in subsequent operations on the CPS to identify the initialization. When this handle is not required, the association can be closed using the `clCkptLibraryFinalize()` and all the allocated resources are freed.

Checkpoint Management

An application can create a Checkpoint by using the `clCkptLibraryCkptCreate()` function. When a Checkpoint is created, the datasets within the Checkpoint can be created using the `clCkptLibraryCkptDataSetCreate()` function. The data is stored and retrieved from these datasets. While creating a dataset, the user has to specify the `serializer` and `deserializer` functions that contain the logic for packing and unpacking the Checkpointed information, respectively. CPS does not interpret the Checkpointed information.
The existence of datasets can be checked by using `clCkptLibraryDoesDatasetExist()` function. The corresponding function for checking the existence of a Checkpoint is `clCkptLibraryDoesCkptExist()`.
An element in a dataset is created using the `clCkptLibraryCkptElementCreate()` function. The user has to specify the element `serialiser` and `deserializer` functions for packing and unpacking the element form a Checkpoint. A dataset is deleted using the `clCkptLibraryCkptDataSetDelete()` function and a Checkpoint can be deleting using the `clCkptLibraryCkptDelete()`.

Data access

CPS provides `clCkptLibraryCkptDataSetWrite()` function to write the entire dataset into the database. An element is to be written to a dataset using the `clCkptLibraryCkptElementWrite()` function. CPS invokes the corresponding `serializer` function and provides a buffer into which the user has to pack the information to be Checkpointed. CPS then stores this buffer into the database.
The contents of a dataset can be read using the `clCkptLibraryCkptDataSetRead()` function. CPS invokes the corresponding `deserializer` function and provides a buffer containing the data that was stored earlier. There is no provision for reading a particular element in a dataset.

Persistence

CPS stores the Checkpointed data in the main memory using database configured by the user in `clDbalConfig.xml`.

2.2.2 Server Based Checkpointing

Checkpoint Life cycle

To use the services of CPS, the application has to initialize the CPS using the `clCkptInitialize()` function. The `clCkptInitialize()` function returns a handle, `ckptServiceHandle`, that can be used in subsequent operations on the CPS to identify the initialization. When this handle is not required, the association can be closed using the `clCkptFinalize()` function .

CPS provides selection object based approach for handling pending callbacks. Applications can use the `clCkptSelectionObjectGet()` and `clCkptDispatch()` functions for the same.

Checkpoint Management

After the CPS is initialized, the application can open a Checkpoint in create/read/write mode using the `clCkptCheckpointOpen()` or `clCkptCheckpointOpenAsync()` functions. A handle, `CheckpointHandle`, that identifies this Checkpoint is returned and the other Checkpoint management and data access functions can use this handle for accessing the Checkpoint.

The applications that have opened the Checkpoint in read/write mode can close the Checkpoint using the `clCkptCheckpointClose()` function. If the Checkpoint is not required and the `clCkptCheckpointDelete()` function is not called, the retention timer (specified during the Checkpoint open) is started. When the timer expires, the Checkpoint is deleted. This is performed to avoid the accumulation of unused Checkpoints in the system. The application can update the retention time of the Checkpoint using the `clCkptCheckpointRetentionDurationSet()` function.

Checkpoints can be deleted using the `clCkptCheckpointDelete()` function. But the Checkpointed data is retained till it is in use. For an asynchronous collocated Checkpoint, the decision where the active replica should reside, lies with the user application. The application can set a particular replica as an active replica using the `clCkptActiveReplicaSet()` function. The user application can also inquire the status (various attributes) of the Checkpoint using the `clCkptCheckpointStatusGet()` function.

Section management

A Checkpoint can have one or more sections. By default, CPS creates a section (called as default section) for every Checkpoint. The information to be Checkpointed is stored in these sections. Associated with each section is a section identifier that identifies a particular section. The user application can create a section in a Checkpoint using the `clCkptSectionCreate()` function and delete a section using the `clCkptSectionDelete()` function.

A section created by the user application is deleted automatically after section expiration time (that is passed in `clCkptSectionCreate()`) is captured. This expiry time can be modified using the `clCkptSectionExpirationTimeSet()` function. The user can also iterate through the sections of a Checkpoint to search for infinite expiry time, expiry time greater than a particular value, or for a corrupted section.

`ClCkptSectionIterationInitialize()`, `clCkptSectionIterationNext()`, and `clCkptSectionIterationFinalize()` functions provide this facility.

Data Access

CPS uses `ioVectors` to read and write data to a Checkpoint. CPS provides two functions

to write data: `clCkptCheckpointWrite()` for writing to multiple sections in a Checkpoint and `clCkptSectionOverwrite()` for updating a particular section of a Checkpoint.
The data can be read from a Checkpoint using the `clCkptCheckpointRead()` function.
CPS does not ensure consistency of data across replicas of asynchronous Checkpoints. The Checkpoint data at replicas can be synchronized using the `clCkptCheckpointSynchronize()` or `clCkptCheckpointSynchronizeAsync()` functions.
CPS also provides notification of immediate consumption to the users. To receive this notification, users can register their callback functions using the `clCkptImmediateConsumptionRegister()` function. The callback is invoked when a change in Checkpointed data occurs.

Replica management

CPS ensures that one more replica of a Checkpoint, in the system provided resource exists. If only the System Director is present, the Checkpoint is replicated in persistent memory. CPS marks the local replica (where the Checkpoint is opened in create mode) as the active replica for all Checkpoints other than asynchronous collocated Checkpoints.

Chapter 3

Service APIs

3.1 Library Based Checkpointing Type Definitions

3.1.1 ClCkptSerializeT

*typedef ClRcT (*ClCkptSerializeT)(*
ClUint32T dsId,
*ClAddrT *pBuffer,*
*ClUint32T *pLen,*
ClHandleT cookie);

The signature of `ClCkptSerializeT` function used to encode the data to be Checkpointed. The parameters of this function are:

- *ClUint32T* - ID of the dataset.
- *ClAddrT* - Address for encoded buffer.
- *ClUint32T* - Length of the encoded buffer.
- *ClHandleT* - Cookie.

3.1.2 ClCkptDeserializeT

*typedef ClRcT (*ClCkptDeserializeT)(*
ClUint32T dsId,
*ClAddrT *pBuffer,*
*ClUint32T *pLen,*
ClHandleT cookie);

The signature of `ClCkptDeserializeT` function used to decode the Checkpointed data.

3.2 Server Based Checkpointing Type Definitions

3.2.1 ClCkptSvcHdlT

typedef ClHandleT ClCkptSvcHdlT;

The type of the handle of the Checkpoint Service instance.

3.2.2 ClCkptSecItrHdlT

typedef ClHandleT ClCkptSecItrHdlT;

The type of the handle for an iteration.

3.2.3 ClCkptHdlT

typedef ClHandleT ClCkptHdlT;

The type of the handle of a Checkpoint.

3.2.4 ClCkptCreationFlagsT

typedef ClUint32T ClCkptCreationFlagsT;

Flags to indicate various attributes of Checkpoint when it is created. The various values can be:

- *CL_CKPT_WR_ALL_REPLICAS* - For synchronous Checkpoint.
- *CL_CKPT_WR_ACTIVE_REPLICA* - For asynchronous Checkpoint.
- *CL_CKPT_WR_ACTIVE_REPLICA_WEAK* - For weak replica.
- *CL_CKPT_WR_CHECKPOINT_COLLOCATED* - For collocated Checkpoint.

3.2.5 ClCkptOpenFlagsT

typedef ClUint32T ClCkptOpenFlagsT;

Flags to indicate open mode such as read, write, or create. The various values can be:

- *CL_CKPT_CHECKPOINT_READ*
- *CL_CKPT_CHECKPOINT_WRITE*
- *CL_CKPT_CHECKPOINT_CREATE*

3.2.6 ClCkptSelectionObjT

typedef ClUint32T ClCkptSelectionObjT;

The type of the handle of a selection object.

3.2.7 ClCkptCallbacksT

typedef struct {
ClCkptCheckpointOpenCallbackT CheckpointOpenCallback;
ClCkptCheckpointSynchronizeCallbackT CheckpointSynchronizeCallback;
} ClCkptCallbacksT;

The structure, `ClCkptCallbacksT`, represents the callback structure provided by the application to the Checkpoint Service. This structure contains the callback functions that the Checkpointing service can invoke. They are:

- *CheckpointOpenCallback* - The type of an asynchronous open callback.
- *CheckpointSynchronizeCallback* - The type of a synchronous open callback.

3.2.8 ClCkptCheckpointCreationAttributesT

typedef struct {
ClSizeT CheckpointSize;
ClCkptCreationFlagsT creationFlags;
ClSizeT maxSectionIdSize;
ClUint32T maxSections;
ClUint32T maxSectionSize;
ClTimeT retentionDuration;
} ClCkptCheckpointCreationAttributesT;

The structure, `ClCkptCheckpointCreationAttributesT`, contains the properties of a Checkpoint that can be specified, when it is created. The attributes of the structure are:

- *CheckpointSize* - Total size of application data in a replica.
- *creationFlags* - Create time attributes.
- *maxSectionIdSize* - Maximum length of the section identifier.
- *maxSections* - Maximum sections for this Checkpoint.
- *maxSectionSize* - Maximum size of a section.
- *retentionDuration* - Duration or period of retention. The Checkpoint that remains inactive for this duration is deleted.

3.2.9 ClCkptCheckpointDescriptorT

typedef struct {
ClCkptCheckpointCreationAttributesT CheckpointCreationAttributes;
ClUint32T memoryUsed;
ClUint32T numberOfSections;
} ClCkptCheckpointDescriptorT;

The structure, `ClCkptCheckpointDescriptorT`, describes a Checkpoint. The attributes of the structure are:

- *CheckpointCreationAttributes* - Creates the attributes.
- *memoryUsed* - Memory used by the Checkpoint.
- *numberOfSections* - Total number of sections.

3.2.10 ClCkptSectionCreationAttributesT

typedef struct {
ClTimeT expirationTime;
*ClCkptSectionIdT *sectionId;*
} ClCkptSectionCreationAttributesT;

The structure, `ClCkptSectionCreationAttributesT`, contains the section attributes that can be specified during the creation process.

- *expirationTime* - Expiration time of the section.
- *sectionId* - Section identifier.

3.2.11 ClCkptSectionIdT

typedef struct {
*ClUint8T *id;*
ClUint16T idLen;
} ClCkptSectionIdT;

The structure, `ClCkptSectionIdT`, contains a section identifier. The attributes of the structure are:

- *id* - Section identifier.
- *idLen* - Length of the section identifier.

3.2.12 ClCkptSectionStateT

typedef enum {
CL_CKPT_SECTION_VALID = 1,
CL_CKPT_SECTION_CORRUPTED = 2
} ClCkptSectionStateT;

The enumeration, `ClCkptSectionStateT`, represents the state of a section in a replica. A section can either be valid or corrupted.

3.2.13 ClCkptSectionDescriptorT

```
typedef struct {
        ClCkptSectionIdT sectionId;
        ClTimeT expirationTime;
        ClSizeT sectionSize;
        ClCkptSectionStateT sectionState;
        ClTimeT lastUpdate;
} ClCkptSectionDescriptorT;
```

This structure, `ClCkptSectionDescriptorT`, represents a section in a Checkpoint. The attributes of the structure are:

- *sectionId* - Section identifier.
- *expirationTime* - Expiration time for the section.
- *sectionSize* - Size of the section.
- *SectionState* - Indicates if a section has a valid or an invalid state.
- *lastUpdate* - Last time the section was updated.

3.2.14 ClCkptSectionsChosenT

```
typedef enum {
        CL_CKPT_SECTIONS_FOREVER,
        CL_CKPT_SECTIONS_LEQ_EXPIRATION_TIME,
        CL_CKPT_SECTIONS_GEQ_EXPIRATION_TIME,
        CL_CKPT_SECTIONS_CORRUPTED,
        CL_CKPT_SECTIONS_ANY
} ClCkptSectionsChosenT;
```

The enumeration, `ClCkptSectionsChosenT`, refers to the selection of sections while iterating through all the sections.

3.2.15 ClCkptIOVectorElementT

```
typedef struct {
        ClPtrT dataBuffer;
        ClOffsetT dataOffset;
        ClSizeT dataSize;
        ClSizeT readSize;
        ClCkptSectionIdT sectionId;
} ClCkptIOVectorElementT;
```

The structure, `ClCkptIOVectorElementT`, contains an IO vector that is used for handling one or more sections. The attributes of the structure are:

- *dataBuffer* - Pointer to the data.
- *dataOffset* - Offset of the data.
- *dataSize* - Size of the data.
- *readSize* - Number of bytes read.
- *sectionId* - Identifier to the section.

3.2.16 ClCkptNotificationCallbackT

*typedef ClRcT(*ClCkptNotificationCallbackT)(ClNameT *pName,ClCkptHdlT ckptHdl);*

The type of the Checkpoint notification callback function that is called when a change in the data of the specified Checkpoint occurs.

3.2.17 ClCkptCheckpointOpenCallbackT

*typedef void (*ClCkptCheckpointOpenCallbackT)(*
ClInvocationT invocation,
ClCkptHdlT CheckpointHandle„
ClRcT error);

Applications (which open a Checkpoint asynchronously) can register with CPS using this callback function. This callback enables them to obtain the status of the asynchronous open.

3.2.18 ClCkptCheckpointSynchronizeCallbackT

*typedef void (*ClCkptCheckpointSynchronizeCallbackT)(*
ClInvocationT invocation,
ClRcT error);

Applications that use asynchronous Checkpoint option are notified about the asynchronous write or open status using this callback. If a problem occurs with the asynchronous write, the application requests Checkpointing service to synchronize all the replicas. This is an asynchronous call. So, the applications can register an optional callback which enables them to know the status of 'synchronize-all-replicas' call.

3.3 Library based Checkpointing Life Cycle APIs

3.3.1 clCkptLibraryInitialize

clCkptLibraryInitialize

Synopsis:
Initializes library based Checkpointing for the invoking application.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptLibraryInitialize( CL_INOUT ClCkptSvcHdlT *pCkptHdl );
```

Parameters:
pCkptHdl: (in/out) Pointer to the handle that identifies the particular initialized instance of Checkpointing library.

Return values:
CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `pCkptHdl` is a NULL pointer.

CL_ERR_NO_MEMORY: Memory allocation failure.

Description:
This function is used to initialize the CPS client and allocates resources to it. The function returns a handle that associates this particular initialization of the Checkpoint library. This handle must be passed as the first input parameter for all functions related to this library.

Library File:
ClCkpt

Related Function(s):
clCkptLibraryFinalize.

3.3.2 clCkptLibraryFinalize

clCkptLibraryFinalize

Synopsis:
Finalizes the library based Checkpointing instance identified by the handle (input parameter).

Header File:
clCkptApi.h

Syntax:
```
ClRcT clCkptLibraryFinalize(CL_IN ClCkptSvcHdlT ckptHdl);
```

Parameters:
ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

Return values:
CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

Description:
This function is used to close the association with Checkpoint Service client. It must be invoked when the Checkpointing services are not required. This invocation frees all related resources allocated during initialization of the Checkpoint library.

Library File:
ClCkpt

Related Function(s):
clCkptInitialize

3.3.3 clCkptLibraryCkptCreate

clCkptLibraryCkptCreate

Synopsis:
Creates a library based Checkpoint.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptCreate(
                                CL_IN ClCkptSvcHdlT  ckptHdl,
                                CL_IN ClNameT       *pCkptName);
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` is a NULL pointer.

CL_ERR_NO_MEMORY: Memory allocation failure.

Description:
This function is used to create a library based Checkpoint. It must be invoked before any further operations can be performed with the Checkpoint.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptDelete.

3.3.4 clCkptLibraryCkptDelete

clCkptLibraryCkptDelete

Synopsis:
Deletes the library based Checkpoint, identified by the name of the Checkpoint (input parameter).

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptDelete(
                              CL_IN ClCkptSvcHdlT ckptHdl,
                              CL_IN ClNameT     *pCkptName);
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint to be deleted.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` is a NULL pointer.

Description:
This function is used to delete the Checkpoint. This must be invoked when the Checkpoint services are not required. This invocation frees all resources associated with the Checkpoint.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptCreate.

3.3.5 clCkptLibraryCkptDataSetCreate

clCkptLibraryCkptDataSetCreate

Synopsis:
Creates a dataset in the Checkpoint and associates it with the `serializer` and `deserializer` functions.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptDataSetCreate(
                            CL_IN ClCkptSvcHdlT ckptHdl,
                            CL_IN ClNameT *pCkptName,
                            CL_IN ClUint32T  dsId,
                            CL_IN ClUint32T grpId,
                            CL_IN ClUint32T  order,
                            CL_IN ClCkptSerializeT dsSerialiser,
                            CL_IN ClCkptDeserializeT dsDeserialiser);
```

Parameters:
ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) A unique identifier for identifying the dataset.

grpId: (in) An optional group ID to group different datasets together. (This is not used in the current implementation.)

order: (in) All the members of a group ID are Checkpointed in the order in which this particular dataset is Checkpointed. (This is not used in the current implementation.)

dsSerialiser: (in) Serialiser/encoder for the data.

dsDeserialiser: (in) Deserialiser/decoder for the data.

Return values:
CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` is a NULL pointer.

CL_ERR_INVALID_PARAMETER: An invalid parameter has been passed to this function. A parameter is set incorrectly.

CL_ERR_NO_MEMORY: Memory allocation failure.

Description:
This function is used to create a dataset and allocate resources to it. This function should be invoked before any further operations can be performed on the dataset. The information in the dataset can then be read, or written into the database.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptDataSetDelete, clCkptLibraryCkptDataSetWrite, clCkptLibraryCkptDataSetRead.

3.3.6 clCkptLibraryCkptDataSetDelete

clCkptLibraryCkptDataSetDelete

Synopsis:
Deletes the dataset from the Checkpoint.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptDataSetDelete(
                                CL_IN ClCkptSvcHdlT ckptHdl,
                                CL_IN ClNameT *pCkptName,
                                CL_IN ClUint32T dsId );
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) Identifier of the dataset to be deleted.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` contains a NULL pointer.

CL_ERR_INVALID_PARAMETER: An invalid parameter has been passed to this function. A parameter is set incorrectly.

CL_ERR_NOT_EXIST: The specified argument does not exist.

Description:
This function is used to delete the dataset from the Checkpoint, that is created using the `clCkptLibraryCkptDataSetCreate()` function.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptDataSetCreate, clCkptLibraryCkptDataSetWrite, clCkptLibraryCkptDataSetRead.

3.3.7 clCkptLibraryCkptDataSetWrite

clCkptLibraryCkptDataSetWrite

Synopsis:
Writes the dataset information into the Checkpoint.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptDataSetWrite(
                                CL_IN ClCkptSvcHdlT ckptHdl,
                                CL_IN ClNameT *pCkptName,
                                CL_IN ClUint32T dsId,
                                CL_IN ClHandleT cookie );
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) Identifier of the dataset to be written.

cookie: (in) User-data, which is opaquely passed to the `serializer` function.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` contains a NULL pointer.

CL_ERR_NOT_EXIST: The specified argument does not exist.

Description:
This function is used to write the dataset information into the Checkpoint. The information is encoded by the `serializer` associated with the dataset.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptDataSetCreate, clCkptLibraryCkptDataSetWrite, clCkptLibraryCkptDataSetRead.

3.3.8 clCkptLibraryCkptDataSetRead

clCkptLibraryCkptDataSetRead

Synopsis:
Reads the stored dataset information from the Checkpoint.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptDataSetRead(
                                CL_IN ClCkptSvcHdlT ckptHdl,
                                CL_IN ClNameT *pCkptName,
                                CL_IN ClUint32T dsId,
                                CL_IN ClHandleT cookie );
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) Identifier of the dataset to be read.

cookie: (in) User-data, which is opaquely passed to the `deserializer`.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` contains a NULL pointer.

CL_ERR_NOT_EXIST: The specified argument does not exist.

Description:
This function is used to read the user information stored in the dataset from Checkpoint. The information is decoded by the `deserializer` associated with the dataset.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptDataSetCreate, clCkptLibraryCkptDataSetWrite, clCkptLibraryCkptDataSetRead.

3.3.9 clCkptLibraryDoesCkptExist

clCkptLibraryDoesCkptExist

Synopsis:
Checks for the existence of a Checkpoint.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryDoesCkptExist(
                                CL_IN ClCkptSvcHdlT   ckptHdl,
                                CL_IN ClNameT       *pCkptName,
                                CL_OUT ClBoolT       *pRetVal);
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

pRetVal: (out) Pointer to the return value. It returns `CL_TRUE`, if the Checkpoint exists.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` or `pRetVal` contains a NULL pointer.

Description:
This function is used to check if a given Checkpoint exists.

Library File:
clCkpt

Related Function(s):
clCkptLibraryDoesDatasetExist.

3.3.10 clCkptLibraryDoesDatasetExist

clCkptLibraryDoesDatasetExist

Synopsis:
Checks the existence of the dataset in the Checkpoint.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryDoesDatasetExist(
                                CL_IN ClCkptSvcHdlT   ckptHdl,
                                CL_IN  ClNameT       *pCkptName,
                                CL_IN  ClUint32T      dsId,
                                CL_OUT ClBoolT       *pRetVal);
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) Identifier of the dataset.

pRetVal: (out) Pointer to the return value. It returns `CL_TRUE`, if the dataset exists.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` or `pRetVal` contains a NULL pointer.

Description:
This function is used to check if a given dataset, identified by `dsId` exists.

Library File:
clCkpt

Related Function(s):
clCkptLibraryDoesCkptExist.

3.3.11 clCkptLibraryCkptElementCreate

clCkptLibraryCkptElementCreate

Synopsis:
Creates an element in the dataset of a Checkpoint and associates it with the `serializer` and `deserializer` functions.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptElementCreate(
                              CL_IN ClCkptSvcHdlT   ckptHdl,
                              CL_IN ClNameT             *pCkptName,
                              CL_IN ClUint32T            dsId,
                              CL_IN ClCkptSerializeT     elemSerialiser,
                              CL_IN ClCkptDeserializeT   elemDeserialiser);
```

Parameters:
ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) Identifier of the dataset.

elemSerialiser: (in) Encoder of the element data.

elemDeserialiser: (in) Decoder of the element data.

Return values:
CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` contains a NULL pointer.

CL_ERR_NOT_EXIST: The specified argument does not exist.

Description:
This function is used to create an element of the dataset. It must be invoked before any element of the dataset can be written or read.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptElementWrite.

3.3.12 clCkptLibraryCkptElementWrite

clCkptLibraryCkptElementWrite

Synopsis:
Writes the user-data to the element of the dataset.

Header File:
clCkptExtApi.h

Syntax:

```
ClRcT clCkptLibraryCkptElementWrite(
                              CL_IN ClCkptSvcHdlT   ckptHdl,
                              CL_IN ClNameT       *pCkptName,
                              CL_IN ClUint32T      dsId,
                              CL_IN ClPtrT         elemId,
                              CL_IN ClUint32T      elemLen,
                              CL_IN ClHandleT      cookie );
```

Parameters:

ckptHdl: (in) Handle to the client, obtained from the `clCkptLibraryInitialize()` function, that identifies this particular initialization of the Checkpoint library.

pCkptName: (in) Pointer to the name of the Checkpoint.

dsId: (in) Identifier of the dataset.

elemName: (in) Identifier of the element.

elemLen: (in) Length of the `elemId`.

cookie: (in) User-data, that is opaquely passed to the `serializer` function.

Return values:

CL_OK: The function executed successfully.

CL_ERR_INVALID_HANDLE: `ckptHdl` is an invalid handle.

CL_ERR_NULL_POINTER: `pCkptName` is a NULL pointer.

CL_ERR_INVALID_PARAMETER: An invalid parameter has been passed to this function. A parameter is set incorrectly.

CL_ERR_NOT_EXIST: The specified argument does not exist.

Description:
This function is used to write the user-data into the element of dataset. Before this function is invoked, the element must be created using the `clCkptLibraryCkptElementCreate()` function. The user-defined `serializer` function will be invoked for the purpose of packing the information.

Library File:
clCkpt

Related Function(s):
clCkptLibraryCkptElementCreate.

3.4 Server based Checkpointing Life Cycle APIs

3.4.1 clCkptInitialize

clCkptInitialize

Synopsis:
Initializes the Checkpoint Service client and registers the various callbacks.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptInitialize(
                                CL_OUT ClCkptSvcHdlT *ckptSvcHandle,
                                CL_IN  const ClCkptCallbacksT *callbacks,
                                CL_INOUT ClVersionT *version);
```

Parameters:

ckptSvcHandle: (out) Checkpoint service handle created by the Checkpoint client and returned to the application. This handle designates this particular initialization of the Checkpoint Service. The application must not modify or interpret this.

callbacks: (in) Optional callbacks for applications that use asynchronous Checkpoints.

version: (in/out) As an input parameter, `version` is a pointer to the required Checkpoint Service version. As an output parameter, the version supported by the Checkpoint Service is delivered.

Return values:

CL_OK: The function executed successfully.

Description:
This function initializes the Checkpoint library for the calling process and registers the various callback functions. This function must be invoked before any other functions of the Checkpoint Service can be used. 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.

Library File:
ClCkpt

Related Function(s):
clCkptFinalize.

3.4.2 clCkptLibraryInitialize

clCkptSelectionObjectGet

Synopsis:
Detects pending callbacks.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSelectionObjectGet(
                    CL_IN ClCkptSvcHdlT ckptHandle,
                    CL_OUT ClSelectionObjectT *pSelectionObject);
```

Parameters:

ckptHandle: (in) The handle obtained through the `clCkptInitialize()` function, that identifies this particular initialization of the Checkpoint Service.

pSelectionObject: (out) A pointer to the operating system handle, that the invoking component/application can use, to detect pending callbacks.

Return values:

CL_OK: The function completed successfully.

CL_CKPT_ERR_INIT_NOT_DONE: On initialization failure.

CL_CKPT_ERR_BAD_HANDLE: `ckptHdl` is an invalid handle.

CL_CKPT_INTERNAL_ERROR: An unexpected problem occurred within the Checkpointing Manager.

CL_CKPT_ERR_INVALID_PARAM: An invalid parameter has been passed to the function. A parameter is not set correctly.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

Description:
This function returns the operating system handle, `pSelectionObject`, associated with the handle `ckptHandle`. The invoking component/application can use this handle to detect pending callbacks, instead of repeatedly invoking `clCkptDispatch()` for this purpose. *pSelectionObject*, returned by this function, is a file descriptor that can be used with `poll()` or `select()` systems call to detect incoming callbacks. It is valid until `clCkptFinalize()` is invoked on the same `ckptHandle` handle.

Library File:
ClCkpt

Related Function(s):
clCkptDispatch.

3.4.3 clCkptDispatch

clCkptDispatch

Synopsis:
Invokes the pending callback in context of the component/application.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptDispatch(
                    CL_IN clCkptSvcHdlT ckptHandle,
                    CL_IN ClDispatchFlagsT dispatchFlags);
```

Parameters:

ckptHandle: (in) The handle obtained through the `clCkptInitialize()` function, that identifies this particular initialization of the Checkpoint Service.

dispatchFlags: (in) Flags that specify the callback execution behavior of the `clCkptDispatch()` function, that contains the values `CL_DISPATCH_ONE`, `CL_DISPATCH_ALL`, or `CL_DISPATCH_BLOCKING`, as defined in `clCommon.h`.

Return values:

CL_OK: The function completed successfully.

CL_CKPT_ERR_INIT_NOT_DONE: The Checkpointing library is not initialized.

CL_CKPT_ERR_BAD_HANDLE: `ckptHdl` is an invalid handle.

CL_CKPT_INTERNAL_ERROR: An unexpected problem occurred within the Checkpointing Manager.

CL_CKPT_ERR_INVALID_PARAM: An invalid parameter has been passed to the function. A parameter is not set correctly.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

Description:
This function invokes, in the context of the calling component/application, pending callbacks for the handle, `ckptHandle`, as specified by the `dispatchFlags` parameter.

Library File:
ClCkpt

Related Function(s):
clCkptSelectionObjectGet.

3.4.4 clCkptFinalize

clCkptFinalize

Synopsis:
Closes the Checkpoint Service client and cancels all pending callbacks related to the handle.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptFinalize(
                                CL_IN ClCkptSvcHdlT ckptHandle);
```

Parameters:
ckptHandle: (in) The handle obtained through the `clCkptInitialize()` function, that identifies this particular initialization of the Checkpoint Service.

Return values:
CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

Description:
The `clCkptFinalize()` function closes the association (represented by the `ckptHandle` parameter,) between the invoking process and the Checkpoint Service. The process must have invoked `clCkptInitialize()` before it can invoke this function. The process must invoke this function once, for each handle acquired, using the `clCkptInitialize()` function.
If the `clCkptFinalize()` function returns successfully, the `clCkptFinalize()` function releases all resources acquired from `clCkptInitialize()` function. It closes all Checkpoints that are open for the particular handle and cancels all pending callbacks related to the particular handle.
After `clCkptFinalize()` is executed, the selection object is no longer valid. As the callback invocation is asynchronous, some callbacks can be processed after this call returns successfully.

Library File:
ClCkpt

Related Function(s):
clCkptInitialize.

3.5 Server based Checkpointing Functional APIs

3.5.1 clCkptCheckpointOpen

clCkptCheckpointOpen

Synopsis:

Opens an existing Checkpoint. If there is no existing Checkpoint, this function creates a new Checkpoint and opens it.

Header File:

clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointOpen(
                              CL_IN ClCkptSvcHdlT ckptHandle,
                              CL_IN const ClNameT *ckeckpointName,
                              CL_IN const ClCkptCheckpointCreationAttributesT
                              *CheckpointCreationAttributes,
                              CL_IN ClCkptOpenFlagsT CheckpointOpenFlags,
                              CL_IN ClTimeT  timeout,
                              CL_OUT ClCkptHdlT *CheckpointHandle);
```

Parameters:

ckptHandle: (in) The handle obtained through the `clCkptInitialize()` function, that identifies this particular initialization of the Checkpoint Service.

ckeckpointName: (in) Name of the Checkpoint to be opened.

CheckpointCreationAttributes: (in) A pointer to the create attributes of a Checkpoint. Refer to `ClCkptCheckpointCreationAttributesT` structure, in the Type Definitions chapter. This parameter must be populated only when the `CREATE` flag is set.

CheckpointOpenFlags: (in) Flags that indicate the required mode in which the Checkpoint needs to be opened. It can have the following values:

- `CL_CKPT_CHECKPOINT_READ`
- `CL_CKPT_CHECKPOINT_WRITE`
- `CL_CKPT_CHECKPOINT_CREATE`

timeout: (in) `clCkptCheckpointOpen()` can fail, if it is not completed within the timeout period. A Checkpoint replica may still be created. This is not supported in the current implementation.

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 the `CheckpointHandle`. The process uses this handle to access the Checkpoint in subsequent invocations of the functions of Checkpoint Service.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `ckeckpointName`, `CheckpointCreationAttributes`, or `CheckpointHandle` contains a NULL pointer.

CL_ERR_ALREADY_EXIST: The Checkpoint already exists.

CL_ERR_NO_MEMORY: There is not enough memory available.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_BAD_FLAG: The flags are incorrect.

CL_CKPT_ERR_INVALID_PARAM: One of the following conditions is true:

- `CheckpointSize > maxSections*maxSectionSize`
- If `CL_CKPT_CHECKPOINT_CREATE` flag is not set and creation attributes are not NULL.
- If `CL_CKPT_CHECKPOINT_CREATE` flag is set and creation attributes are NULL.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

Description:

The `clCkptCheckpointOpen()` function opens a Checkpoint. If the Checkpoint does not exist and, if `CL_CKPT_CHECKPOINT_CREATE` flag is set, the Checkpoint is created.
The invocation to this function is blocking and a new Checkpoint handle is returned when the function is executed successfully. A Checkpoint can be opened multiple times for reading and/or writing in the same process or in different processes.
A combination of the creation flags, (defined in `ClCkptCheckpointCreationFlagsT`,) are bitwise ORed together to provide the value of the `creationFlags` field of the `CheckpointCreationAttributes` parameter.

Library File:

ClCkpt

Related Function(s):

clCkptCheckpointOpenAsync, clCkptCheckpointClose.

3.5.2 clCkptCheckpointOpenAsync

clCkptCheckpointOpenAsync

Synopsis:
Creates and opens a Checkpoint asynchronously.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointOpenAsync(
                              CL_IN ClCkptSvcHdlT ckptHandle,
                              CL_IN ClInvocationT invocation,
                              CL_IN const ClNameT  *CheckpointName,
                              CL_IN const ClCkptCheckpointCreationAttributesT
                              *CheckpointCreationAttributes,
                              CL_IN ClCkptOpenFlagsT CheckpointOpenFlags);
```

Parameters:

ckptHandle: (in) The handle obtained through the `clCkptInitialize()` function, that identifies this particular initialization of the Checkpoint Service.

invocation: (in) (in) This parameter is used by the application to identify the callback.

CheckpointName: (in) Name of the Checkpoint to be opened.

CheckpointCreationAttributes: (in) Pointer to the create attributes of a the Checkpoint. Refer to `ClCkptCheckpointCreationAttributesT` structure.

CheckpointOpenFlags: (in) Flags to indicate the desired mode to open. It can have the following values:

- `CL_CKPT_CHECKPOINT_READ`
- `CL_CKPT_CHECKPOINT_WRITE`
- `CL_CKPT_CHECKPOINT_CREATE`

Return values:

CL_OK: The function executed successfully.

CL_CKPT_ERR_BAD_FLAG: The flags are incorrect.

CL_CKPT_ERR_INVALID_PARAM: One of the following conditions is true:

- `CheckpointSize > maxSections*maxSectionSize`
- If `CL_CKPT_CHECKPOINT_CREATE` flag is not set and creation attributes are not NULL.
- If `CL_CKPT_CHECKPOINT_CREATE` flag is set and creation attributes are NULL.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

Description:
This function is used to create and open a Checkpoint asynchronously.

Library File:
ClCkpt

Related Function(s):
clCkptCheckpointOpen, clCkptCheckpointClose.

3.5.3 clCkptCheckpointClose

clCkptCheckpointClose

Synopsis:
Closes the Checkpoint designated by the `CheckpointHandle`.

Header File:
clCkptApi.h

Syntax:
```
ClRcT clCkptCheckpointClose(
                                CL_IN ClCkptHdlT CheckpointHandle);
```

Parameters:
CheckpointHandle: (in) The handle, `CheckpointHandle`, obtained from the `clCkptCheckpointOpen()` function.

Return values:
CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

Description:
This function closes the Checkpoint, identified by `CheckpointHandle`, opened using the `clCkptCheckpointOpen()` or `clCkptCheckpointOpenAsync()` functions.
After this function is executed, the handle, `CheckpointHandle`, becomes invalid. The deletion of a Checkpoint frees all resources allocated by the Checkpoint Service. When a process is terminated, the opened Checkpoints of the process are closed. This call cancels all pending callbacks that refer directly or indirectly to the handle, `CheckpointHandle`.
If `clCkptCheckpointDelete()` has already been called, then by calling the `clCkptCheckpointClose()` function, the reference count to this Checkpoint becomes zero and the Checkpoint is deleted. After this call, if the reference count becomes zero, and `clCkptCheckpointDelete()` has not been called, then the retention timer associated with the Checkpoint (provided as part of `clCkptCheckpointOpenAsync()` or `clCkptCheckpointOpen()` functions) is started. When the timer expires, the Checkpoint is deleted.

Library File:
ClCkpt

Related Function(s):
clCkptCheckpointOpen, clCkptCheckpointOpenAsync, clCkptCheckpointDelete.

3.5.4 clCkptCheckpointDelete

clCkptCheckpointDelete

Synopsis:
Removes the Checkpoint from the system and frees all resources allocated to it.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointDelete(
                              CL_IN ClCkptSvcHdlT ckptHandle,
                              CL_IN const ClNameT  *CheckpointName);
```

Parameters:

ckptHandle: (in) The handle obtained through the `clCkptInitialize()` function, that identifies this particular initialization of the Checkpoint Service.

CheckpointName: (in) Pointer to the name of the Checkpoint that needs to be deleted.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `CheckpointName` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_INUSE: The Checkpoint is already in use.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_INVALID_PARAM: An invalid parameter has been passed to the function. A parameter is not set correctly.

Description:

This function deletes an existing Checkpoint, identified by `CheckpointName` from the cluster. After completion of this function:

- The name `CheckpointName` becomes invalid. Any invocation of a Checkpoint Service function that uses the name of the Checkpoint returns an error, unless a Checkpoint is re-created with this name. The Checkpoint can be re-created by specifying the same name of the Checkpoint to be unlinked in an open call with the `CL_CKPT_CHECKPOINT_CREATE` flag set. This way, a new instance of the Checkpoint is created, while the old instance of the Checkpoint is not yet deleted.

  Note that this is similar to the way POSIX treats files.

- If no process has the Checkpoint open when `clCkptCheckpointDelete()` is invoked, the Checkpoint is immediately deleted.

- Any process that has the Checkpoint open can still continue to access it. The Checkpoint is deleted when the last `clCkptCheckpointClose()` operation is performed.

  The deletion of a Checkpoint frees all resources allocated by the Checkpoint Service. This function can be invoked by any process, and the invoking process need not be the creator or opener of the Checkpoint.

Library File:
ClCkpt

Related Function(s):
clCkptCheckpointOpen, clCkptCheckpointOpenAsync, clCkptCheckpointClose.

3.5.5 clCkptCheckpointRetentionDurationSet

clCkptCheckpointRetentionDurationSet

Synopsis:
Sets the retention duration of a Checkpoint.

Header File:
clCkptApi.h

Syntax:
```
ClRcT clCkptCheckpointRetentionDurationSet(
                              CL_IN ClCkptHdlT CheckpointHandle,
                              CL_IN ClTimeT retentionDuration);
```

Parameters:
CheckpointHandle: (in) The Checkpoint whose retention time is being set. This handle should be obtained using the `clCkptCheckpointOpen()` function.

retentionDuration: (in) Duration for which the Checkpoint can be retained.

Return values:
CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_BAD_OPERATION: The Checkpoint has already been un-linked.

Description:
This function is used to set the retention duration of a Checkpoint. If reference count becomes zero by executing the `clCkptCheckpointClose()` function, and `clCkptCheckpointDelete()` has not been called, the retention timer associated with the Checkpoint (provided by `clCkptCheckpointOpenAsync()` or `clCkptCheckpointOpen()` functions) is started. When the timer expires, the Checkpoint is deleted.

Library File:
ClCkpt

Related Function(s):
None.

3.5.6 clCkptActiveReplicaSet

clCkptActiveReplicaSet

Synopsis:
Sets the local replica as the active replica.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptActiveReplicaSet(
                                CL_IN ClCkptHdlT CheckpointHandle);
```

Parameters:
CheckpointHandle: (in) The handle of the Checkpoint obtained using the `clCkptCheckpointOpen()` function.

Return values:
CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_BAD_OPERATION: The Checkpoint has already been un-linked.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in write mode.

Description:
This function is used to set the local replica to be active replica, if no active replica is set for the Checkpoint.

Library File:
ClCkpt

Related Function(s):
None.

3.5.7 clCkptCheckpointStatusGet

clCkptCheckpointStatusGet

Synopsis:
Returns the status and the attributes of the Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointStatusGet(
                                CL_IN ClCkptHdlT CheckpointHandle,
                                CL_OUT ClCkptCheckpointDescriptorT *CheckpointStatus);
```

Parameters:

CheckpointHandle: (in) The handle of the Checkpoint obtained using the `clCkptCheckpointOpen()` function.

CheckpointStatus: (out) Pointer to Checkpoint descriptor in the address space of the invoking process. This contains the Checkpoint status information to be returned. Refer to `ClCkptCheckpointDescriptorT` structure, in the Type Definitions chapter.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: The `CheckpointStatus` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

Description:

This function is used to return the status and the various attributes of the Checkpoint. The list of attributes are defined by the `ClCkptCheckpointDescriptorT` structure. This function retrieves the `CheckpointStatus` of the Checkpoint identified by `CheckpointHandle`.
If the Checkpoint was created using either `CL_CKPT_WR_ACTIVE_REPLICA` or `CL_CKPT_WR_ACTIVE_REPLICA_WEAK` option, the Checkpoint status is obtained from the active replica. If the Checkpoint was created using the `CL_CKPT_WR_ALL_REPLICAS` option, the Checkpoint service determines the replica from which the Checkpoint status can be obtained.

Library File:
ClCkpt

Related Function(s):
None.

3.5.8 clCkptSectionCreate

clCkptSectionCreate

Synopsis:
Creates a section in the Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSectionCreate(
                              CL_IN ClCkptHdlT CheckpointHandle,
                              CL_IN ClCkptSectionCreationAttributesT
                              *sectionCreationAttributes,
                              CL_IN const ClUint8T *initialData,
                              CL_IN ClSizeT initialDataSize);
```

Parameters:

CheckpointHandle: (in) The handle obtained using the `clCkptCheckpointOpen()` function.

sectionCreationAttributes: (in) Pointer to the structure, `ClCkptSectionCreationAttributesT`, that contains the `in/out` field `sectionId` and the `in` field `expirationTime`.

initialData: (in) Pointer to the location in the address space of the invoking process that contains the initial data of the section to be created.

initialDataSize: (in) Size in bytes of the initial data of the section to be created. The maximum initial size is indicated by `maxSectionSize`, as specified by the Checkpoint creation attributes in `clCkptCheckpointOpen()` function.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `sectionCreationAttributes` or `initialData` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_ALREADY_EXIST: The section, defined in section creation attributes, already exists or the Checkpoint was created to have only one section.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_INVALID_PARAMETER: One of the following conditions is true:

- `initialDataSize` is zero, but `initialData` is not NULL
- `initialDataSize > maxSectionSize`
- `sectionId` length is > `maxSectionId` length

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in write mode.

CL_CKPT_ERR_NO_SPACE: The maximum number of sections is reached.

Description:

This function creates a new section in the Checkpoint, identified by `CheckpointHandle` as long as the total number of existing sections is less than the maximum number of sections specified in a call to `clCkptCheckpointOpen()` or `clCkptCheckpointOpenAsync()` functions. Unlike a Checkpoint, a section need not be opened for access. The section is deleted by the Checkpoint Service when its expiration time is reached.
If a Checkpoint is created with only one section, it is not necessary to create that section. The default section is identified by the special identifier, `CL_CKPT_DEFAULT_SECTION_ID`.
If the Checkpoint is created with the `CL_CKPT_WR_ALL_REPLICAS` property, the section is created in all the Checkpoint replicas. Otherwise, the section is created in the active Checkpoint replica, and is created asynchronously in the other Checkpoint replicas.

Library File:

ClCkpt

Related Function(s):

clCkptSectionDelete, clCkptSectionExpirationTimeSet, clCkptSectionIterationInitialize, clCkptSectionIterationNext, clCkptSectionIterationFinalize, clCkptCheckpointWrite, clCkptSectionOverwrite.

3.5.9 clCkptSectionDelete

clCkptSectionDelete

Synopsis:
Deletes a section in the given Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSectionDelete(
                                CL_IN ClCkptHdlT CheckpointHandle,
                                CL_IN const ClCkptSectionIdT *sectionId);
```

Parameters:

CheckpointHandle: (in) The handle to the Checkpoint that contains the section to be deleted, obtained from the `clCkptCheckpointOpen()` function.

sectionId: (in) A pointer to the identifier of the section to be deleted.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `sectionId` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

CL_CKPT_ERR_INVALID_PARAMETER: An invalid parameter has been passed to the function. A parameter is not set correctly.

Description:
This function deletes a section in the Checkpoint identified by `CheckpointHandle`. If the Checkpoint is created with the `CL_CKPT_WR_ALL_REPLICAS` property, the section is deleted in all the Checkpoint replicas. Otherwise, the section is deleted in at least the active Checkpoint replica. The default section, identified by `CL_CKPT_DEFAULT_SECTION_ID`, cannot be deleted using the `clCkptSectionDelete()` function.

Library File:
ClCkpt

Related Function(s):
clCkptSectionCreate, clCkptSectionExpirationTimeSet, clCkptSectionIterationInitialize, clCkptSectionIterationNext, clCkptSectionIterationFinalize, clCkptCheckpointWrite, clCkptSectionOverwrite.

3.5.10 clCkptSectionExpirationTimeSet

clCkptSectionExpirationTimeSet

Synopsis:
Sets the expiration time of a section.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSectionExpirationTimeSet(
                                CL_IN ClCkptHdlT CheckpointHandle,
                                CL_IN const ClCkptSectionIdT*   sectionId,
                                CL_IN ClTimeT  expirationTime);
```

Parameters:

CheckpointHandle: (in) The handle obtained using the `clCkptCheckpointOpen()` function.

sectionId: (in) Identifier to the section.

expirationTime: (in) Expiration time of the section.

Return values:

CL_OK: The function executed successfully.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

CL_CKPT_ERR_INVALID_PARAMETER: An invalid parameter has been passed to the function. A parameter is not set correctly.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

Description:
This function is used to set the expiration time of a section.

Library File:
ClCkpt

Related Function(s):
clCkptSectionCreate, clCkptSectionDelete, clCkptSectionIterationInitialize, clCkptSectionIterationNext, clCkptSectionIterationFinalize, clCkptCheckpointWrite, clCkptSectionOverwrite.

3.5.11 clCkptSectionIterationInitialize

clCkptSectionIterationInitialize

Synopsis:
Allows the application to iterate through the sections of a Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSectionIterationInitialize(
                                CL_IN ClCkptHdlT CheckpointHandle,
                                CL_IN  ClCkptSectionsChosenT   sectionsChosen,
                                CL_IN ClTimeT  expirationTime,
                                CL_OUT ClHandleT *sectionIterationHandle);
```

Parameters:

CheckpointHandle: (in) The handle obtained using the `clCkptCheckpointOpen()` function.

sectionsChosen: Rule for the iteration. Refer the `ClCkptSectionChosenT` enumeration in the Type Definitions chapter.

expirationTime: (in) Expiration time used along with the rule.

sectionIterationHandle: (out) Handle used to identify the current section.

Return values:

CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_ERR_NO_MEMORY: Memory allocation failure.

Description:
This function is used to enable the application to iterate through the sections in a Checkpoint.

Library File:
ClCkpt

Related Function(s):
clCkptSectionCreate, clCkptSectionDelete, clCkptSectionExpirationTimeSet, clCkptSectionIterationNext, clCkptSectionIterationFinalize, clCkptCheckpointWrite, clCkptSectionOverwrite.

3.5.12 clCkptSectionIterationNext

clCkptSectionIterationNext

Synopsis:
Returns the next section in the list of sections.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSectionIterationNext(
                    CL_IN  ClHandleT   sectionIterationHandle,
                    CL_OUT ClCkptSectionDescriptorT *sectionDescriptor);
```

Parameters:

sectionIterationHandle: (in) Handle to the Checkpoint obtained from the `clCkptCheckpointOpen()` function.

sectionDescriptor: (out) Pointer to the descriptor of a section. Refer the `ClCkptSectionDescriptorT` structure, in the Type Definitions chapter.

Return values:

CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_ERR_NULL_POINTER: `sectionDescriptor` contains a NULL pointer.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

Description:
This function is used to retrieve the next section in the list of sections matching the `ClCkptSectionIterationInitialize()` call.

Library File:
ClCkpt

Related Function(s):
clCkptSectionCreate, clCkptSectionDelete, clCkptSectionExpirationTimeSet, clCkptSectionIterationInitialize, clCkptSectionIterationFinalize, clCkptCheckpointWrite, clCkptSectionOverwrite.

3.5.13 clCkptSectionIterationFinalize

clCkptSectionIterationFinalize

Synopsis:
Frees resources associated with the iteration.

Header File:
clCkptApi.h

Syntax:
```
ClRcT clCkptSectionIterationFinalize(
                    CL_IN ClHandleT sectionIterationHandle);
```

Parameters:
sectionIterationHandle: (in) Handle of the iteration.

Return values:
CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

Description:
This function is used to free the resources associated with the iteration identified by the `sectionIterationHandle`.

Library File:
ClCkpt

Related Function(s):
clCkptSectionCreate, clCkptSectionDelete, clCkptSectionExpirationTimeSet, clCkptSectionIterationInitialize, clCkptSectionIterationNext, clCkptCheckpointWrite, clCkptSectionOverwrite.

3.5.14 clCkptCheckpointWrite

clCkptCheckpointWrite

Synopsis:
Writes multiple sections to a given Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointWrite(
                    CL_IN  ClCkptHdlT  CheckpointHandle,
                    CL_IN  const ClCkptIOVectorElementT   *ioVector,
                    CL_IN  ClUint32T  numberOfElements,
                    CL_OUT ClUint32T  *erroneousVectorIndex);
```

Parameters:

CheckpointHandle: (in) The handle to the Checkpoint obtained using the `clCkptCheckpointOpen()` or `clCkptCheckpointOpenAsync()` functions.

ioVector: (in) Pointer to the `ioVector` containing the section IDs and the data to be written.

numberOfElements: (in) Total number of elements in `ioVector`.

erroneousVectorIndex: (out) Pointer to the index (first element of `iovector`), stored in the address space of the caller, that makes the invocation fail. If the index is set to NULL, or if the invocation succeeds, the field remains unchanged. This is updated, if the `ClCkptCheckpointWrite()` function fails.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `erroneousVectorIndex` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

Description:

This function writes data from the memory regions specified by `ioVector` into a Checkpoint:

- If this Checkpoint is created with the `CL_CKPT_WR_ALL_REPLICAS` property, all of the Checkpoint replicas are updated. If the function call does not complete or returns with an error, no data is written.
- If the Checkpoint is created with the `CL_CKPT_WR_ACTIVE_REPLICA` property, the active Checkpoint replica is updated. Other Checkpoint replicas are updated asynchronously. If the invocation does not complete or returns with an error, no data is written.

- If the Checkpoint is created with the `CL_CKPT_WR_ACTIVE_REPLICA_WEAK` property, the active Checkpoint replica is updated. Other Checkpoint replicas are updated asynchronously. If the invocation returns with an error, no data is written. 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 function call, several sections and several portions of sections can be updated simultaneously. The elements of the `ioVectors` are written in the order, `ioVector`[0] to `ioVector` [numberOfElements - 1]. As a result of this function call, some sections may grow in size.

Library File:

ClCkpt

Related Function(s):

clCkptSectionCreate, clCkptSectionDelete, clCkptSectionExpirationTimeSet, clCkptSectionIterationInitialize, clCkptSectionIterationNext, clCkptSectionIterationFinalize, clCkptSectionOverwrite.

3.5.15 clCkptSectionOverwrite

clCkptSectionOverwrite

Synopsis:
Writes a single section into a given Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptSectionOverwrite(
                    CL_IN  ClCkptHdlT  CheckpointHandle,
                    CL_IN const ClCkptSectionIdT *sectionId,
                    CL_IN const void  *dataBuffer,
                    CL_IN ClSizeT dataSize);
```

Parameters:

CheckpointHandle: The handle, identifying the Checkpoint, obtained from `clCkptCheckpointOpen()` or `clCkptCheckpointOpenAsync()` functions.

sectionId: (in) Pointer to an identifier for the section that is to be overwritten. If this pointer points to `CL_CKPT_DEFAULT_SECTION_ID`, the default section is updated.

dataBuffer: (in) Pointer to the buffer from where the data is being copied.

dataSize: (in) Size in bytes of the data to be written. This becomes the new size for this section.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `sectionId` or `dataBuffer` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

Description:
This function is used to write a single section into a given Checkpoint. This function is similar to `ClCkptCheckpointWrite()` function except that it overwrites only a single section. As a result of this invocation, the previous data and size of the section changes. This function may be used, even if there was no prior call to the `ClCkptCheckpointWrite()` function.

Library File:
ClCkpt

Related Function(s):
clCkptSectionCreate, clCkptSectionDelete, clCkptSectionExpirationTimeSet, clCkptSectionIterationInitialize, clCkptSectionIterationNext, clCkptSectionIterationFinalize, clCkptCheckpointWrite.

3.5.16 clCkptCheckpointRead

clCkptCheckpointRead

Synopsis:
Reads multiple sections at a time.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointRead(
                    CL_IN  ClCkptHdlT  CheckpointHandle,
                    CL_INOUT ClCkptIOVectorElementT   *ioVector,
                    CL_IN ClUint32T  numberOfElements,
                    CL_OUT ClUint32T  *erroneousVectorIndex);
```

Parameters:

CheckpointHandle: (in) The handle, identifying the Checkpoint, obtained from `clCkptCheckpointOpen()` or `clCkptCheckpointOpenAsync()` functions.

ioVector: (in/out) Pointer to the IO Vector containing the section IDs and the data to be written.

numberOfElements: (in) Total number of elements in `ioVector`. Every element is of type, `ClCkptIOVectorElementT`, and contains the following fields:

- sectionId: Identifier of the section to be read from.
- dataBuffer: (in/out) Pointer to the buffer containing the data to be read. If `dataBuffer` is NULL, the value of `datasize` provided by the caller is ignored and the buffer is provided by the Checkpoint Service library. The buffer must be de-allocated by the caller.
- dataSize: Size of the data to be read to the buffer, identified by `dataBuffer`. The maximum size is indicated by `maxSectionSize`, as specified in the creation attributes of the Checkpoint.
- dataOffset: Offset of the section that marks the start of the data that is to be read.
- readSize: (out) Used by `clCkptCheckpointRead()` to record the number of bytes of data that have been read.

erroneousVectorIndex: (out) Pointer to the index for errors in `ioVector` (first vector element). This is an index in address space of the caller, that causes the function call to fail. If the function call succeeds, `erroneousVectorIndex` is set to `NULL` and should be ignored.

Return values:

CL_OK: The function executed successfully.

CL_ERR_NULL_POINTER: `ioVector` or `erroneousVectorIndex` contains a NULL pointer.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_ERR_NO_MEMORY: Memory allocation failure.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

CL_CKPT_ERR_INVALID_PARAMETER: One of the following conditions is true:

- `offset > maxSecSize`
- Section boundary is violated

Library File:
ClCkpt

Description:
This function is used to read multiple sections at a time. It can also be used to read a single section.

Related Function(s):
None.

3.5.17 clCkptCheckpointSynchronize

clCkptCheckpointSynchronize

Synopsis:
Synchronizes the replicas of a Checkpoint.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointSynchronize(
                    CL_IN  ClCkptHdlT  CheckpointHandle,
                    CL_IN ClTimeT      timeout);
```

Parameters:

CheckpointHandle: (in) Handle to the Checkpoint obtained from the `clCkptCheckpointOpen()` function.

timeout: (in) Timeout to execute the operation. This is not supported in the current implementation.

Return values:

CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

CL_CKPT_ERR_BAD_OPERATION: This Checkpoint is not an asynchronous Checkpoint.

Description:
This function is used to synchronize the replicas of a Checkpoint.

Library File:
ClCkpt

Related Function(s):
clCkptCheckpointSynchronizeAsync.

3.5.18 clCkptCheckpointSynchronizeAsync

clCkptCheckpointSynchronizeAsync

Synopsis:
Synchronizes the replicas of a Checkpoint asynchronously.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptCheckpointSynchronizeAsync(
                    CL_IN  ClCkptHdlT  CheckpointHandle,
                    CL_IN ClInvocationT invocation);
```

Parameters:

CheckpointHandle: (in) Handle to the Checkpoint obtained from the `clCkptCheckpointOpen()` function.

invocation: (in) Identifies this call when the callback function is invoked.

Return values:

CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_VERSION_MISMATCH: The client and server versions are incompatible.

CL_CKPT_ERR_NOT_EXIST: No active replica exists.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

CL_CKPT_ERR_OP_NOT_PERMITTED: The Checkpoint is not opened in read mode.

CL_CKPT_ERR_BAD_OPERATION: This Checkpoint is not an asynchronous Checkpoint.

Description:
This function is used to synchronize the replicas of a Checkpoint asynchronously.

Library File:
ClCkpt

Related Function(s):
clCkptCheckpointSynchronize.

3.5.19 clCkptImmediateConsumptionRegister

clCkptImmediateConsumptionRegister

Synopsis:
Registers a callback function to be called to notify change in the Checkpoint data.

Header File:
clCkptApi.h

Syntax:

```
ClRcT clCkptImmediateConsumptionRegister(
                    CL_IN  ClCkptHdlT  CheckpointHandle,
                    CL_IN ClCkptNotificationCallbackT callback);
```

Parameters:
CheckpointHandle: (in) Handle to the Checkpoint obtained from the `clCkptCheckpointOpen()` function.

invocation: (in) Identifies this call when the callback function is invoked.

Return values:
CL_OK: The function executed successfully.

CL_CKPT_ERR_NOT_INITIALIZED: Checkpoint library is not initialized.

CL_CKPT_ERR_INALID_HANDLE: `CheckpointHandle` is an invalid handle.

Description:
This function is used to register a callback function that is called, when the specified Checkpoint data is updated.

Library File:
ClCkpt

Related Function(s):
None.

Chapter 4

Service Management Information Model

TBD

Chapter 5

Service Notifications

TBD

Chapter 6

Configuration

TBD

Chapter 7

Debug CLIs

TBD

Glossary

Glossary of OpenClovis Checkpoint Service Terms

Checkpoint A checkpoint is an entity used by applications to store their states and related information, so that the same can be retrieved in a failover, switchover, or restart scenario.

Dataset Each checkpoint is structured as datasets in File/Library-based Checkpointing. A dataset is a part of checkpoint, that can be modified or read independently from other datasets. It is similar to sections, used in Server-based Checkpointing.

Serializer A user-defined function that is called to pack the user-data before it is stored in the persistent memory.

Deserializer A user-defined function that unpacks the stored data. It is invoked when data is read from persistent memory.

Element Elements are data of similar types stored within a dataset.

Sections Each checkpoint is structured as sections in Server-based Checkpointing. A section is a part of a checkpoint, that can be modified/read independently from other sections.

Synchronous checkpoint When a checkpoint is created with synchronous update option (Synchronous checkpoint), all the `writes` and checkpoint management calls return only after the checkpoint replicas are updated.

Asynchronous checkpoint When a checkpoint is created with asynchronous update option (Asynchronous checkpoint), all the `writes` and checkpoint management calls return after the active replicas of the checkpoint are updated. Other replicas are updated asynchronously.

Asynchronous non-collocated checkpoint The type of an asynchronous checkpoint, where the active replica is selected by CPS.

Asynchronous collocated checkpoint The type of an asynchronous checkpoint, where the active replica is selected by the user.

Replica/Checkpoint replica A copy of the data that is stored in a checkpoint.

Active replica The replica that is updated first or read from an asynchronous checkpoint. There can be a maximum of one active replica at any given time.

Local replica A replica located on the node where the checkpoint is opened.

Synchronization The process of synchronizing the data of a checkpoint across all replicas.

Retention duration Duration for which a checkpoint is retained after the users have closed it (or users are not using it).

Section expiry time Duration after which the section is deleted and becomes unuseable.

Default section If the maximum sections specified by the user is equal to 1, when the checkpoint is created, CPS provides this section by default. The expiry time of this default section is infinite.

Section identifier The identifier of the section that is unique within a checkpoint.

Section iteration The process of iterating through all the sections of a checkpoint, to find sections that match a criteria specified by the user.

Index