Centera

Version 3.1

API REFERENCE GUIDE

P/N 069001185
REV A07

EMC Corporation
Corporate Headquarters:
Hopkinton, MA 01748-9103
1-508-435-1000
www.emc.com

Published January 2006

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

This Software Development Kit (SDK) contains the intellectual property of EMC Corporation or is licensed to EMC Corporation from third parties. Use of this SDK and the intellectual property contained therein is expressly limited to the terms and conditions of the License Agreement.

License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this documentation and/or software.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes software developed by L2FProd.com (http://www.L2FProd.com/).

Trademark Information

IRIX® and SGI® are registered trademarks of Silicon Graphics, inc. in the United States and other countries worldwide.

PostScript and Adobe are trademarks of Adobe Systems Incorporated.

All other trademarks used herein are the property of their respective owners.

ReiserFS is hereby licensed under the GNU General Public License version 2.

Source code files that contain the phrase "licensing governed by reiserfs/README" are "governed files" throughout this file. Governed files are licensed under the GPL. The portions of them owned by Hans Reiser, or authorized to be licensed by him, have been in the past, and likely will be in the future, licensed to other parties under other licenses. If you add your code to governed files, and don't want it to be owned by Hans Reiser, put your copyright label on that code so the poor blight and his customers can keep things straight. All portions of governed files not labeled otherwise are owned by Hans Reiser, and by adding your code to it, widely distributing it to others or sending us a patch, and leaving the sentence in stating that licensing is governed by the statement in this file, you accept this. It will be a kindness if you identify whether Hans Reiser is allowed to license code labeled as owned by you on your behalf other than under the GPL, because he wants to know if it is okay to do so and put a check in the mail to you (for non-trivial improvements) when he makes his next sale. He makes no guarantees as to the amount if any, though he feels motivated to motivate contributors, and you can surely discuss this with him before or after contributing. You have the right to decline to allow him to license your code contribution other than under the GPL.

Further licensing options are available for commercial and/or other interests directly from Hans Reiser: hans@reiser.to. If you interpret the GPL as not allowing those additional licensing options, you read it wrongly, and Richard Stallman agrees with me, when carefully read you can see that those restrictions on additional terms do not apply to the owner of the copyright, and my interpretation of this shall govern for this license.

Finally, nothing in this license shall be interpreted to allow you to fail to fairly credit me, or to remove my credits, without my permission, unless you are an end user not redistributing to others. If you have doubts about how to properly do that, or about what is fair, ask. (Last I spoke with him Richard was contemplating how best to address the fair crediting issue in the next GPL version.)

Contents

Preface

This guide is part of the EMC Centera Software Development Kit (SDK), and is for experienced programmers who are developing applications that interface with a Centera cluster. It is intended to be a complete reference guide for both C and Java application development using the Centera API.

Here is an overview of where information is located in this manual.

- Chapter 1, *C API Reference*, is a reference guide to the C API.
- Appendix A, *Monitoring Information*, contains information on the Monitoring API.
- Appendix B, *Deprecated Functions*, lists the deprecated functions.

Related Documentation

Other Centera publications include:

- *Centera Quick Start Guide*, P/N 300-002-546
- *Centera Programmer's Guide*, P/N 069001127
- *Centera Online Help*, P/N 300-002-656
- *Cabinet Setup Guide for the 40U Cabinet*, P/N 014003099
- *Site Preparation and Unpacking Guide for the 40U Cabinet*, P/N 014003100

Conventions Used in this Manual

EMC uses the following conventions for notes and cautions.

Note: A note presents information that is important, but not hazard-related.

CAUTION

A caution contains information essential to avoid damage to the system or equipment. The caution may apply to hardware or software.

Typographical Conventions

EMC uses the following type style conventions in this guide:

AVANT GARDE	Keystrokes
Palatino, bold	◆ Dialog box, button, icon, and menu items in text ◆ Selections you can make from the user interface, including buttons, icons, options, and field names
Palatino, italic	◆ New terms or unique word usage in text ◆ Command line arguments when used in text ◆ Book titles
`Courier, italic`	Arguments used in examples of command line syntax.

`Courier`	System prompts and displays and specific filenames or complete paths. For example:
	`working root directory [/user/emc]:`
	`c:\Program Files\EMC\Symapi\db`

`Courier, bold`	◆ User entry. For example: **`symmpoll -p`** ◆ Options in command line syntax

Where to Get Help

For questions about technical support and service, contact your service provider.

If you have a valid EMC service contract, contact the EMC Customer Support Center at:

United States:	**(800) 782-4362 (SVC-4EMC)**
Canada:	**(800) 543-4782 (543-4SVC)**
Worldwide:	**(508) 497-7901**

Follow the voice menu prompts to open a service call, then select Centera Product Support.

Sales and Customer Service Contacts

For the list of EMC sales locations, please access the EMC home page at:

http://emc.com/contact/

For additional information on the EMC products and services available to customers and partners, refer to the EMC Powerlink Web site at:

http://powerlink.emc.com

Your Comments

Your suggestions will help us continue to improve the accuracy, organization, and overall quality of the user publications. Please send a message to **techpub_comments@emc.com** with your opinions of this guide.

Your technical enhancement suggestions for future development consideration are welcome. To send a suggestion, log on to **http://powerlink.emc.com**, follow the path **Support**, **Contact Support**, and choose **Software Product Enhancement Request** from the Subject menu.

1

C API Reference

This chapter is a reference guide for application developers who are working with the Centera™ Access C API.

The main sections in this chapter are:

Function Syntax

This section details the syntax of Centera Access API function calls.

Function Call

Function names consist of a prefix followed by the actual function. The prefix refers to the object type on which the function operates.

The API functions are divided into the following categories:

- **Pool** functions — prefix is **FPPool**
- **Clip** functions — prefix is **FPClip or FPClipID**
- **Tag** functions — prefix is **FPTag**
- **Retention Class** functions — prefix is **FPRetentionClassContext** or **FPRetentionClass**
- **Stream** functions — prefix is **FPStream**
- **Query** functions:
  - **Query Expression** functions — prefix is **FPQueryExpression**
  - **Pool Query** functions — prefix is **FPPoolQuery**
  - **Query Result** functions — prefix is **FPQueryResult**
- **Monitoring** functions — prefix is **FPMonitor** or **FPEventCallback**
- **Time Format** functions — prefix is **FPTime**

Example `FPPool_Open()` opens a pool, `FPClip_Open()` opens a C-Clip.

Parameter List

The parameter list contains all parameters that a function requires. Each parameter is preceded by its type definition and is prefixed by one of the following:

- "in" — Input parameter
- "out" — Output parameter
- "io" — Input and output parameter

Commas separate parameters in the parameter list:

```
(parameter_type parameter1, parameter_type parameter2, …)
```

Some parameter types are API-specific. Refer to *API Data Types* on page 1-7.

Example `FPPoolRef inPool` is a reference to a pool.

The `void` type is used for functions that do not require a parameter or have no return value:

```
FPPool_GetLastError (void)
```

Unicode and Wide Character Support

The Centera API supports Unicode characters (ISO-10646) for specific calls that read, write or update C-Clip metadata, specifically UTF-8, UTF-16 and UTF-32 encodings. On architectures that support 2 byte wide characters, the Centera API also supports UCS-2 encodings.

Functions that accept string arguments have a wide character and three Unicode variants. Each variant has a suffix indicating its type of string support.

For example, the function to open a pool (`FPPool_Open (const char *inPoolAddr)`) accepts a string argument (the connection string). Therefore, the `FPPool_Open` function has the following wide character and Unicode variants:

```
FPPool_OpenW (const wchar_t *inPoolAddr)
FPPool_Open8 (const FPChar8 *inPoolAddr)
FPPool_Open16 (const FPChar16 *inPoolAddr)
FPPool_Open32 (const FPChar32 *inPoolAddr)
```

Note: A separate reference page is not provided for each variant function, since aside from the function name and data type, the calls are all the same.

If you are localizing your application, use the "8", "16", and "32" functions. Otherwise, you can use the default Latin-1 (no suffix) and wide ("W") functions. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information.

Table 1-1 lists the function variants.

Table 1-1 Function Variants

Function Suffix	String Parameter Type	Character Width (bits)	Encoding
None	char *	8	Latin-1
W	wchar_t *	16 or 32 (platform dependent)	UCS2 (for 16-bit characters) or UCS4 (for 32-bit characters)
8	char *	8	UTF-8
16	FPChar16 *	16	UTF-16
32	FPChar32 *	32	UTF-32

Note: Centera API calls that retrieve metadata (for example, `FPClip_GetDescriptionAttribute`) may have parameters that contain the address of an integer value that specifies the size of a user-provided buffer in this way: as an input parameter that sizes the user supplied buffer which the API may overwrite with the size that the output text actually requires.

This integer value is determined by the data type that the API call is using.

For example, if the `ioAttrValueLen` parameter of the `FPClip_GetDescriptionAttribute16` routine is set to 20 by the user, then a buffer of 40 bytes would be provided. After the call, the value of `ioAttrValueLen` may be set to 10 by the API, indicating that 20 bytes was actually needed to store the UTF-16 string into `outAttrValue`.

The following table lists the input and output units for API calls that deal with string encoding:

API String-encoded Calls	Input and Output Units
Latin-1	Number of char
Wide	Number of wchar_t
UTF-8	Number of char
UTF-16	Number of FPChar16
UTF-32	Number of FPChar32

Note: A UTF-8 encoded character may take up 6 bytes, while a UTF-16 encoded character may take up 4 bytes. Therefore, care should be taken when allocating UTF-8 and UTF-16 related buffers.

Unicode and Wide Character Routines

The following routines support Unicode and wide characters. These characters append to the routine name, for example, `FPPool_OpenW`, `FPPool_Open8`, `FPPool_Open16`, and `FPPool_Open32`.

- `FPClip_AuditedDelete`
- `FPClip_Create`
- `FPClip_GetDescriptionAttribute`
- `FPClip_GetDescriptionAttributeIndex`
- `FPClip_GetEBRClassName`
- `FPClip_GetEBREventTime`
- `FPClip_GetName`
- `FPClip_GetRetentionClassName`
- `FPClip_RemoveDescriptionAttribute`
- `FPClip_SetDescriptionAttribute`
- `FPClip_SetRetentionHold`
- `FPMonitor_Open`
- `FPPool_GetCapability`
- `FPPool_GetClusterTime`
- `FPPool_GetComponentVersion`
- `FPPool_GetGlobalOption`
- `FPPool_GetIntOption`
- `FPPool_Open`
- `FPPool_RegisterApplication`
- `FPPool_SetGlobalOption`
- `FPPool_SetIntOption`
- `FPQueryExpression_DeselectField`
- `FPQueryExpression_IsFieldSelected`
- `FPQueryExpression_SelectField`
- `FPQueryResult_GetField`
- `FPRetentionClass_GetName`
- `FPRetentionClassContext_GetNamedClass`
- `FPTag_Create`
- `FPTag_GetBoolAttribute`
- `FPTag_GetIndexAttribute`
- `FPTag_GetLongAttribute`
- `FPTag_GetStringAttribute`
- `FPTag_GetTagName`
- `FPTag_RemoveAttribute`
- `FPTag_SetBoolAttribute`
- `FPTag_SetLongAttribute`
- `FPTag_SetStringAttribute`
- `FPTime_MillisecondsToString`
- `FPTime_SecondsToString`
- `FPTime_StringToMilliseconds`
- `FPTime_StringToSeconds`

API Data Types

The Centera API uses the following data types:

Data Type	Definition
FPLong	64-bit signed integer
FPInt	32-bit signed integer
FPShort	16-bit signed integer
FPBool	Boolean with possible values `true` (1) and `false` (0)
FPChar16	16-bit character with UTF-16 encoding
FPChar32	32-bit character with UTF-32 encoding
FPClipID	Character array used to identify a C-Clip
FPErrorInfo	Structure that holds error information, which is retrieved by `FPPool_GetLastErrorInfo()`. The application should not deallocate or modify the pointer member variables. The FPErrorInfo structure is detailed in Table 1-2 on page 1-8.
FPPoolInfo	Structure that specifies pool information. The application should not deallocate or modify the pointer member variables. The FPPoolInfo structure is detailed in Table 1-3 on page 1-8.
FPStreamInfo	Stream structure that passes information to and from callback functions. The FPStreamInfo structure is detailed in Table 1-4 on page 1-9.

Table 1-2 FPErrorInfo Structure

FPInt error	The last FPLibrary error that occurred on the current thread.
FPInt systemError	The last system error that occurred on this thread.
char* trace	The function trace for the last error that occurred.
char* message	The message associated with the FPLibrary error.
char* errorString	The error string associated with the FPLibrary error.
FPShort errorClass	The class of message: • FP_NETWORK_ERRORCLASS (network error) • FP_SERVER_ERRORCLASS (Centera cluster error) • FP_CLIENT_ERRORCLASS (client application error, probably due to coding error or wrong usage)

Table 1-3 FPPoolInfo Structure

FPInt poolInfoVersion	The current version of this structure (2).
FPLong capacity	The total capacity of the pool, in bytes.
FPLong freeSpace	The total free usable space of the pool, in bytes.
char clusterID[128]	The cluster identifier of the pool.
char clusterName[128]	The name of the cluster.
char version[128]	The version of the pool server software.
char replicaAddress[256]	A comma-separated list of the replication cluster's node (with the access role) addresses as specified when replication was enabled; empty if replica cluster not identified or configured.

Table 1-4 FPStreamInfo Structure

short mVersion	The current version of FPStreamInfo.
void *mUserData	Application-specific data, untouched by the SDK.
FPLong mStreamPos	The current position in the stream.
FPLong mMarkerPos	The position of the stream marker.
FPLong mStreamLen	The length of the stream in bytes, if known, else –1.
FPBool mAtEOF	True if the end of stream has been reached.
FPBool mReadFlag	Read/write indicator, true on FPTag_BlobWrite(), false on FPTag_BlobRead().
void *mBuffer	The data buffer supplied by the application.
FPLong mTransferLen	The number of bytes to be transferred or actually transferred.

Pool Functions

The pool functions operate at the pool level. A pool consists of one or more Centera clusters, each with its own IP address or DNS name and port number(s).

The application must establish a connection to the pool before performing a pool operation, with the exception of `FPPool_SetGlobalOption()` and `FPPool_GetComponentVersion()`. The application should provide one or more addresses of the available nodes with the access role to make a connection to a pool.

The pool functions are thread safe.

Note: You should close the connection to a pool if that connection is no longer needed, in order to free all used resources. However, it is better to reuse existing pool connections than to frequently open and close connections.

- `FPPool_Close`
- `FPPool_GetCapability`
- `FPPool_GetClipID`
- `FPPool_GetClusterTime`
- `FPPool_GetComponentVersion`
- `FPPool_GetGlobalOption`
- `FPPool_GetIntOption`
- `FPPool_GetLastError`
- `FPPool_GetLastErrorInfo`
- `FPPool_GetPoolInfo`
- `FPPool_GetRetentionClassContext`
- `FPPool_Open`
- `FPPool_RegisterApplication`
- `FPPool_SetClipID`
- `FPPool_SetGlobalOption`
- `FPPool_SetIntOption`

FPPool_Close

Syntax: `FPPool_Close (const FPPoolRef inPool)`

Return Value: `void`

Input Parameters: `const FPPoolRef inPool`

Concurrency Requirement: This function is thread safe.

Description: This function closes the connection to the given pool and frees resources associated with the connection. Note that calling this function on pool that is already closed may produce unwanted results.

Note: Be sure to close all pool connections that are no longer needed in order to avoid performance loss and resource leakage. However, it is better to reuse existing pool connections than to frequently open and close connections.

Parameters: `const FPPoolRef inPool`
The reference to a pool opened by `FPPool_Open()`.

Example:

```
FPPool_Close (myPool);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OBJECTINUSE_ERR` (client error)

FPPool_GetCapability

Syntax:

```
FPPool_GetCapability (const FPPoolRef inPool, const char
   *inCapabilityName, const char
   *inCapabilityAttributeName, char *outCapabilityValue,
   FPInt *ioCapabilityValueLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPPoolRef inPool, const char *inCapabilityName,
const char *inCapabilityAttributeName, FPInt
*ioCapabilityValueLen
```

Output Parameters:

```
char *outCapabilityValue, FPInt *ioCapabilityValueLen
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function returns the attribute value for the given attribute name of a capability. The capabilities are associated with the access profile and refer to the operations that the SDK is allowed to perform on the cluster. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const char *inCapabilityName`
  The name of the capability. Refer to Table 1-5, *Capability Names and Attributes*, for the capability names, attribute names, and attribute values that represent the capabilities.

Table 1-5 Capability Names and Attributes

Capability Name	Attribute Name	Attribute Value	Interpretation
FP_BLOBNAMING	FP_SUPPORTED_SCHEMES	MD5/MG	The supported naming schemes.
FP_CLIPENUMERATION	FP_ALLOWED	true/false	If true, `FPPoolQuery_Open()` is allowed.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.
FP_COMPLIANCE	FP_MODE	basic/ce/ce+	The compliance mode of the cluster being connected to in a pool. Note: The `ce` mode refers to the Governance Edition (GE).
	FP_EVENT_BASED_RETENTION FP_RETENTION_HOLD	supported/ unsupported	If supported, this retention capability allows the SDK to quickly verify an application's Advanced Retention Management license for EBR and retention hold. If unsupported, the SDK rejects the call before the C-Clip is written, and generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`.
	FP_RETENTION_MIN_MAX		It also determines whether the EBR retention period falls within the retention minimum and maximum range. If not within the range and the license is supported, the SDK generates the error `FP_RETENTION_OUT_OF_BOUNDS_ERR`.
FP_DELETE	FP_ALLOWED	true/false	If true, `FPClip_Delete()` and `FPClip_AuditedDelete()` are allowed.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.

Table 1-5 Capability Names and Attributes (continued)

Capability Name	Attribute Name	Attribute Value	Interpretation
FP_DELETIONLOGGING	FP_SUPPORTED	true/false	If true, the server creates reflections when deleting C-Clips.
FP_EXIST	FP_ALLOWED	true/false	If true, `FPClip_Exists()` and `FPTag_BlobExists()` are allowed.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.
FP_MONITOR	FP_ALLOWED	true/false	If true, the server supports the `FPMonitor_xxx` calls.
FP_POOL_POOLMAPPINGS	FP_POOLS	string	The pool mappings for all the profiles or pools.
	FP_PROFILES		The list of profiles with a pool mapping.
FP_PRIVILEGEDDELETE	FP_ALLOWED	true/false	If true, privileged deletion using `FPClip_AuditedDelete()` is allowed. This value is never true for a CE+ model.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.
FP_PURGE	FP_ALLOWED	true/false	Deprecated.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.
FP_READ	FP_ALLOWED	true/false	If true, `FPClip_Open()` and `FPTag_BlobRead() (Partial)` are allowed.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.

Table 1-5 Capability Names and Attributes (continued)

Capability Name	Attribute Name	Attribute Value	Interpretation
FP_RETENTION	FP_DEFAULT	integer	If the CDF does not specify a retention period or retention class, the default value is `FP_NO_RETENTION_PERIOD` (`0`) for a CE mode (GE model), `FP_INFINITE_RETENTION_PERIOD` (`-1`) for a CE+ mode (CE+ model), or `FP_DEFAULT_RETENTION_PERIOD` (`-2`) for a default value. Note: The system administrator can edit the default value only for the CE mode.
	FP_FIXED_RETENTION_MIN	64-bit integer	The minimum time allowed for a fixed retention period. If absent, the default value is 0 for `FP_NO_RETENTION_PERIOD`. This minimum constraint applies to all newly written C-Clips in Centera v3.1.
	FP_FIXED_RETENTION_MAX		The maximum time allowed for a fixed retention period. If absent, the default value is -1 for `FP_INFINITE_RETENTION_PERIOD`. This maximum constraint applies to all newly written C-Clips in Centera v3.1.
	FP_VARIABLE_RETENTION_MIN		The minimum time allowed for an EBR period. If absent, the default value is 0 for `FP_NO_RETENTION_PERIOD`. This minimum constraint applies to all newly written C-Clips in Centera v3.1.
	FP_VARIABLE_RETENTION_MAX		The maximum time allowed for an EBR period. If absent, the default value is -1 for `FP_INFINITE_RETENTION_PERIOD`. This maximum constraint applies to all newly written C-Clips in Centera v3.1.

Table 1-5 Capability Names and Attributes (continued)

Capability Name	Attribute Name	Attribute Value	Interpretation
FP_RETENTION_HOLD	FP_ALLOWED	true/false	If true, retention hold is enabled, which allows the access profile to set and release retention holds at the C-Clip level. This capability also allows the SDK to check if retention hold is allowed. If false, the SDK rejects any retention hold call before the C-Clip is written, and generates the error `FP_OPERATION_NOT_ALLOWED`.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.
FP_WRITE	FP_ALLOWED	true/false	If true, `FPClip_Write()` and `FPTag_BlobWrite()` are allowed.
	FP_POOLS	string	The list of pools associated with the tag capability. Pools display in a comma-separated list. If there are no pools, the string is empty.

- `const char *inCapabilityAttributeName`
  The name of the capability attribute.
- `char *outCapabilityValue`
  `outCapabilityValue` is the memory buffer that receives the value of the capability upon successful completion of the function.
- `FPInt *ioCapabilityValueLen`
  Input: The reserved length, in characters, of the `outCapabilityValue` buffer.
  Output: The actual length of the string, in characters, including the end-of-string character.

Example: Check if the SDK is allowed to perform a privileged delete operation:

```
char vCapability[256];
FPInt vCapabilityLen = sizeof (vCapability);
FPPool_GetCapability(vPool, FP_PRIVILEGEDDELETE,
   FP_ALLOWED, vCapability, &vCapabilityLen);
if (strcmp(vCapability,FP_TRUE) == 0) { ... }
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (program logic error)

FPPool_GetClipID

Syntax

```
void FPPool_GetClipID (const FPPoolRef inPool,
FPClipID outContentAddress)
```

Return Value

`void`

Parameters

- `const FPPoolRef inPool`
  The reference to a pool that has been opened by a call to the `FPPool_Open` function.
- `const FPClipID outContentAddress`
  The buffer that receives the ID of the Profile Clip.

Description

The `FPPool_GetClipID` function retrieves the Profile Clip associated with the access profile on an open pool and places it in the buffer specified by the `outContentAddress` parameter.

A Profile Clip is a Content Address that is associated with an access profile.

An access profile contains information about the identity of a client application and determines the operations that the client application can perform on the cluster. For more information on access profiles, see the *Centera Programmer's Guide,* P/N 069001127.

Note: An access profile can be associated with only one Profile Clip.

Example

For an example of how to call this routine, see the manpage for *Clip Functions* on page 1-47.

Error Handling

The `FPPool_GetClipID` function returns `ENOERR` if successful or the following error codes:

Note: This call will fail from the SDK side if the specified pool has not first been opened, if the Profile Clip has not previously been set by a call to `FPPool_SetClipID`, or if the Profile Clip has been deleted.

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_SDK_INTERNAL_ERR` (program logic error)

FPPool_GetClusterTime

Syntax:

```
FPPool_GetClusterTime (const FPPoolRef inPool, char
   *outClusterTime, FPInt *ioClusterTimeLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPPoolRef inPool, FPInt *ioClusterTimeLen
```

Output Parameters:

```
char *outClusterTime, FPInt *ioClusterTimeLen
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves the current cluster time. The time is specified in UTC (Coordinated Universal Time, also known as GMT —Greenwich Mean Time).

For example, February 21, 2004 is expressed as: `2004.02.21 10:46:32 GMT`

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `char *outClusterTime`
  `outClusterTime` is the memory buffer that will store the cluster time. The time is specified in `YYYY.MM.DD hh:mm:ss` GMT format.
- `FPInt *ioClusterTimeLen`
  Input: The reserved length, in characters, of the `outClusterTime` buffer.
  Output: The actual length of the string, in characters, including the end-of-string character.

Example:

```
char vClusterTime[256];
FPInt vClusterTimeLen=256;
FPPool_GetClusterTime(vPool, vClusterTime,
   &vClusterTimeLen);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)

FPPool_GetComponentVersion

Syntax:
```
FPPool_GetComponentVersion (const FPInt inComponent, char
   *outVersion, FPInt *ioVersionLen)
```

Return Value:
```
void
```

Input Parameters:
```
const FPInt inComponent, FPInt *ioVersionLen
```

Output Parameters:
```
char *outVersion, FPInt *ioVersionLen
```

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function retrieves the version of SDK components that are currently in use. Use `FPPool_GetPoolInfo()` to retrieve the CentraStar® version of the cluster.

Parameters:

- `const FPInt inComponent`
  `inComponent` refers to the component queried for its version. Use one of the following values:

  `FP_VERSION_FPLIBRARY_DLL`

  `FP_VERSION_FPLIBRARY_JAR` (Java only)

- `char *outVersion`
  `outVersion` is the memory buffer that will store the version number.

- `FPInt *ioVersionLen`
  Input: The reserved length, in characters, of the `outVersion` buffer.
  Output: The actual length of the string, in characters, including the end-of-string character.

Example:
```
char vVersion[128];
FPInt vVersionLen;
FPPool_GetComponentVersion(FP_VERSION_FPLIBRARY_DLL,
   vVersion, &vVersionLen);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_UNKNOWN_OPTION` (program logic error)
- `FP_PARAM_ERR` (program logic error)

FPPool_GetGlobalOption

Syntax: `FPPool_GetGlobalOption (const char *inOptionName)`

Return Value: `FPInt`

Input Parameters: `const char *inOptionName`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function returns the value of `inOptionName` that is set by `FPPool_SetGlobalOption()`.

Parameters: `const char *inOptionName`
Refer to *FPPool_SetGlobalOption* on page 1-38 for the option names and their possible values.

Example:
```
FPPool_GetGlobalOption(FP_OPTION_RETRYCOUNT);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_UNKNOWN_OPTION` (program logic error)

FPPool_GetIntOption

Syntax:

```
FPPool_GetIntOption (const FPPoolRef inPool, const char
   *inOptionName)
```

Return Value:

```
FPInt
```

Input Parameters:

```
const FPPoolRef, const char *inOptionName
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function returns the value of `inOptionName` that is set by `FPPool_SetIntOption()`.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const char *inOptionName`
  Refer to *FPPool_SetClipID* on page 1-35 for the option names and their possible values.

Example:

```
FPPool_GetIntOption(myPool,
   FP_OPTION_ENABLE_MULTICLUSTER_FAILOVER);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_UNKNOWN_OPTION` (program logic error)

FPPool_GetLastError

Syntax: `FPPool_GetLastError (void)`

Return Value: `FPInt`

Concurrency Requirement: This function is thread safe.

Description: This function returns the error status of the last FPLibrary function call on the same thread and returns the error number. It is recommended that your application check the error status after each function call. Refer to *Error Codes* on page 1-275 for a complete list of FPLibrary-specific errors. If no error was generated, the return value is `ENOERR` (zero).

Call `FPPool_GetLastErrorInfo()` to retrieve additional information about the error.

Note: If the SDK is unable to return the error status of the last FPLibrary function call, the return value is `FP_UNABLE_TO_GET_LAST_ERROR`. This value indicates that an error was generated by the `FPPool_GetLastError()` call itself and not the previous function call. The error status of the previous function call is unknown and may have succeeded.

Parameters: `void`

Example:

```
FPInt errorCode;

errorCode = FPPool_GetLastError();
if (errorCode != ENOERR)
{
   /* Process the error ... */
}
```

FPPool_GetLastErrorInfo

Syntax:

```
FPPool_GetLastErrorInfo (FPErrorInfo *outErrorInfo)
```

Return Value:

```
void
```

Output Parameters:

```
FPErrorInfo *outErrorInfo
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves the error status of the last FPLibrary function call and returns information about the error in the `FPErrorInfo` structure. The error can be FPLibrary-specific or OS-specific. Refer to Table 1-2 on page 1-8 for details of the `FPErrorInfo` structure.

If no errors are generated, the returned structure has `null` values in its data fields.

Note: Do not modify the contents of the `outErrorInfo` structure. Do not deallocate the string member variables.

Parameters:

`FPErrorInfo *outErrorInfo`
The structure that will store the error information that the function retrieves. Refer to Table 1-2 on page 1-8 for details of the `FPErrorInfo` structure. If no errors are generated, the returned structure has `null` values in its data fields.

Example:

```
FPInt errorCode;
FPErrorInfo errInfo;

errorCode = FPPool_GetLastError();
if (errorCode != ENOERR)
{

   FPPool_GetLastErrorInfo(&errInfo);
   /* Process the Error ...*/
}
```

FPPool_GetPoolInfo

Syntax: `FPPool_GetPoolInfo (const FPPoolRef inPool, FPPoolInfo *outPoolInfo)`

Return Value: `void`

Input Parameters: `const FPPoolRef inPool`

Output Parameters: `FPPoolInfo *outPoolInfo`

Concurrency Requirement: This function is thread safe.

Description: This function retrieves information about the current cluster and saves it into `outPoolInfo`.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `FPPoolInfo *outPoolInfo`
  The structure that will store the pool information that the function retrieves. The structure is defined as follows:
  - `poolInfoVersion [FPInt]`: The current version of this information structure.
  - `capacity [FPLong]`: The total usable capacity (in bytes) of all online nodes in the current cluster.

Note: The CLI and Centera Viewer report 1024 bytes as 1 KB. EMC recommends using the same conversion rate when converting the capacity as returned by the SDK to your application.

  - `freeSpace [FPLong]`: The total free space (in bytes) in the current cluster.

Note: This function returns an approximate value for the free space. The returned values can vary within 2% when compared to subsequent calls of the function. The free space reflects the total amount of usable space on all online nodes to store data mirrored.

  - `clusterID [string]`: The cluster identifier of the current cluster (maximum of 128 characters).
  - `clusterName [string]`: The cluster name of the current cluster (maximum of 128 characters).

- `version [string]`: The version of the CentraStar software on the current cluster (maximum of 128 characters).
- `replicaAddress [string]`: A comma-separated list of the replica cluster's node with the access role addresses as specified when replication was configured. The maximum number of characters is 256. The string is empty if replication is not configured.

Note: All pool information can be set only by using the CLI. Refer to the *Centera Online Help*, P/N 300-002-656, for more information.

Example:

```
FPPoolInfo PoolInfo;
FPPool_GetPoolInfo(myPool, &PoolInfo);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_VERSION_ERR` (internal error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_SERVER_NOT_READY_ERR` (server error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPPool_GetRetentionClassContext

Syntax:

```
FPPool_GetRetentionClassContext (const FPPoolRef
   inPoolRef)
```

Return Value:

`FPRetentionClassContextRef`

Input Parameters:

`const FPPoolRef inPoolRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns the retention class context—the set of all defined retention classes—for the primary cluster of the given pool. This function returns `NULL` and sets the `FP_SERVER_ERR` error status if the cluster does not support retention classes.

Use the `FPRetentionClassContext_xxx()` functions to retrieve individual classes from the retention class context.

Call `FPRetentionClassContext_Close()` when you no longer need the `FPRetentionClassContextRef` object to free all associated (memory) resources.

Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on retention classes.

Parameters:

`const FPPoolRef inPoolRef`
The reference to a pool opened by `FPPool_Open()`.

Example:

```
FPRetentionClassContextRef myRetClassContext;
myRetClassContext = FPPool_GetRetentionClassContext
   (myPool);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OUT_MEMORY_ERR`
- Additional server errors

FPPool_Open

Syntax: `FPPool_Open (const char *inPoolAddress)`

Return Value: `FPPoolRef`

Input Parameters: `const char *inPoolAddress`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function initiates connections to one or more clusters. The pool object manages these connections. This function returns a reference to the opened pool.

The pool address (`inPoolAddress`) is a comma-separated string of IP addresses or DNS names of available nodes with the access role. The pool probes all addresses in this connection string either immediately (normal strategy, the default) or on an as-needed basis (lazy strategy). Refer to *Pool Open Strategy* in the *Centera Programmer's Guide*, P/N 069001127, for more information. You can change the default open strategy; refer to *FPPool_SetGlobalOption* on page 1-38.

Note: If the SDK cannot connect to a cluster—when the connection string includes an inaccessible cluster, or when a cluster has been configured to replicate to an inaccessible cluster—the `FPPool_Open()` call will take at least 1 minute to complete.

You can globally specify how many connections can be made to a pool by calling `FPPool_SetGlobalOption()`. The default value is 100 and the maximum value is 999. Refer to *Connection Pooling* in the *Centera Programmer's Guide*, P/N 069001127, for more information.

Be sure to close all pool connections (*FPPool_Close* on page 1-12) that are no longer needed to free associated resources.

Parameters: `const char *inPoolAddress`
`inPoolAddress` is a comma-separated string containing one or more addresses of the available nodes with the access role of the pool. The format is:

```
pooladdress ::= hintlist
hintlist ::= hint ("," hint)*
hint ::= [ protocol "://" ] ipreference [ ":" port ]
protocol ::= "hpp"
port ::= [0-9]+ (default is 3218)
ipreference ::= dnsname | ip-address
dnsname ::= DNS name is a DNS maintained name that
   resolves to one or more IP addresses (using
   round-robin) max length is 256 chars
ip-address ::= 4-tuple address format
```

A hint is a single pool address and a hintlist contains one or more hints.

Profile Information

You can augment the connection string with the PEA file or username/secret for the PAI module to be used by the application. For example:

```
   "10.2.3.4,10.6.7.8?c:\centera\rwe.pea"
```

or

```
   "10.2.3.4,10.6.7.8?name=<username>,secret=<password>"
```

You also can assign multiple profiles on a connection string to access one or more clusters. For more information on PAI modules and the syntax of connection strings, refer to the *Centera Programmer's Guide*, P/N 069001127.

Connection Failover Prefixes

Addresses prefixed with `primary=`, called primary addresses, are eligible for becoming the primary cluster. Addresses prefixed with `secondary=`, called secondary addresses, are not eligible for becoming the primary cluster. An address without a prefix is a primary address.

For example:

```
   "10.2.3.4,primary=10.6.7.8,secondary=10.11.12.13"
```

Both 10.2.3.4 and 10.6.7.8 are primary addresses, and 10.11.12.13 is a secondary address.

If all primary connections for nodes with the access role fail, the `FPPool_Open()` function fails. The `primary=` and `secondary=` prefixes are case-sensitive, and there can be no whitespace before or after the equal sign (=).

Refer to *Multicluster Failover* in the *Centera Programmer's Guide*, P/N 069001127, for more information on connection failover.

Example: This example specifies a connection string with multiple IP addresses—a best practice that protects against one or more nodes with the access role of a cluster being unavailable.

```
myPool = FPPool_Open
   ("10.1.1.1,10.1.1.2,10.1.1.3,10.1.1.4");
```

This example opens a pool using the specified PEA file.

```
myPoolName = "10.62.69.153?c:\centera\rwe.pea";
myPool = FPPool_Open (myPoolName);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_NO_POOL_ERR` (network error)
- `FP_ACCESSNODE_ERR` (network error)
- `FP_AUTHENTICATION_FAILED_ERR` (server error)

FPPool_RegisterApplication

Syntax:

```
FPPool_RegisterApplication (const char* inAppName,
   const char* inAppVer)
```

Return Value:

```
void
```

Input Parameters:

```
const char* inAppName, const char* inAppVer
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function stores an application's name and version on the Centera log.

Note: To ensure that this application information is written to a Centera log, you must call `FPPool_RegisterApplication` before calling `FPPool_Open()`.

Alternatively, you can use the environment variables `FP_OPTION_APP_NAME` and `FP_OPTION_APP_VER` to log this information without changing existing applications. The use of these environment variables is recommended, as they can improve the performance of `FPPool_Open()`.

Note: Using `FPPool_RegisterApplication` overrides the option settings set as environment variables.

Parameters:

- `const char* inAppName`
  `inAppName` is a string with the name of the application.
- `const char* inAppVer`
  `inAppVer` is a string with the application version.

Example:

```
FPPool_RegisterApplication (inAppName, inAppVer);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPPool_SetClipID

Syntax

```
void FPPool_SetClipID (const FPPoolRef inPool, const
   FPClipID inContentAddress);
```

Return Value

`void`

Parameters

`const FPPoolRef inPool`
Reference to a pool that has been opened by a call to the `FPPool_Open` function.

`const FPClipID inContentAddress`
The Profile Clip to be associated with the access profile. This is the Content Address returned by a prior call to `FPClip_Create`.

Description

The `FPPool_SetClipID` function sets the Profile Clip for the access profile on an open pool.

Before you call `FPPool_SetClipID`, you must first successfully create a C-Clip with the `FPClip_Create` function and write the clip to disk with a call to `FPClip_Write`.

A Profile Clip is a Content Address that is associated with an access profile.

An access profile contains information about the identity of a client application and determines the operations that the client application can perform on the cluster. For more information on access profiles, see the *Centera Programmer's Guide*, P/N 069001127.

Note: An access profile can be associated with only one Profile Clip.

Example

```
FPClipID vID1;
FPClipID vID2;

FPPoolRef myPool = FPPool_Open("10.241.35.101");

// Create clip
FPClipRef vClipRef = FPClip_Create(myPool, "Test");
FPInt vRetVal = print_last_error();

if ( vRetVal == 0 && vClipRef )
{
    //Write the clip with no data

    FPClip_Write(vClipRef, vID1);
    vRetVal = print_last_error();
    if (vRetVal == 0)
    {
        //Set this clip as the profile clip

        FPPool_SetClipID(myPool, vID1);
        vRetVal = print_last_error();
        if (vRetVal == 0)
        {
            //Close this clip

            FPClip_Close(vClipRef);
            vRetVal = print_last_error();
            if (vRetVal == 0)
            {
                //Get the profile clip CA

                FPPool_GetClipID(myPool, vID2);
                vRetVal = print_last_error();
            }
        }
    }
}
```

Error Handling

The `FPPool_SetClipID` function returns `ENOERR` if successful or the following error codes:

Note: This call will fail from the SDK side if the specified pool has not first been opened, if the Profile Clip is not specified, or if the specified Profile Clip is not a viable Content Address.

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PROFILECLIPID_NOTFOUND_ERR` (server error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_SERVER_ERR` (server error)
- `FP_PACKET_FIELD_MISSING_ERR` (internal error)
- `FP_VERSION_ERR` (internal error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_SERVER_NOT_READY_ERR` (server error)
- `FP_UNKNOWN_AUTH_SCHEME_ERR` (server error)
- `FP_UNKNOWN_AUTH_PROTOCOL_ERR` (server error)
- `FP_AUTHENTICATION_FAILED_ERR` (server error)
- `FP_TRANSACTION_FAILED_ERR` (server error)
- `FP_PROFILECLIPID_WRITE_ERR` (client error)

FPPool_SetGlobalOption

Syntax:

```
FPPool_SetGlobalOption (const char *inOptionName, const
   FPInt inOptionValue)
```

Return Value:

```
void
```

Input Parameters:

```
const char *inOptionName, const FPInt inOptionValue
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function sets application-wide options. When set, the new values take effect immediately including all running threads of the application. However, it does not affect other applications using the same FPLibrary.

Note: You can set any global pool option as an environment variable. The option settings that an application sets take precedence and override those that were previously set as environment variables.

Parameters:

- `const char *inOptionName`
  `inOptionName` is a string with the name of the option to be set.

  Global Options

  You can set any of the following global options.

  - `FP_OPTION_CLUSTER_NON_AVAIL_TIME` — The time in seconds that a cluster is marked as not available before retrying with a probe. Other clusters in the pool will be used while the cluster is unavailable. The default value is 600 (10 minutes). The minimum is 0. The maximum is 36000 (10 hours).
  - `FP_OPTION_DISABLE_CLIENT_STREAMING` — Disables the `CLIENT_CALCID_STREAMING` mode and converts it to the `SERVER_CALCID_STREAMING` mode, which allows only the Centera server (not the client) to calculate the content address of the blob data on the server.

- `FP_OPTION_EMBEDDED_DATA_THRESHOLD` — The maximum data size, in bytes, for data to be embedded in the CDF instead of being stored as separate blobs. The default value is 0 bytes, meaning data is never embedded in the CDF. The maximum value is 102400 bytes (100 KB). The value for the embedded data threshold can be set to less than or equal to 102400 bytes.

  Embedding data in the CDF can improve write performance. However, embedded data does not benefit from single-instance storage. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information.

  When the data size is unknown (for example, when using `FP_OPTION_CLIENT_CALCID_STREAMING` and the data size exceeds the prefetch buffer), data is not embedded in the CDF regardless of the threshold.

  You can explicitly control data embedding, which overrides this threshold setting, when you call `FPTag_BlobWrite()`.

- `FP_OPTION_MAXCONNECTIONS` — The maximum number of sockets that the SDK will allocate for your application. Sockets are used to communicate with the Centera clusters managed in each pool object. The default value is 100. The maximum value is 999.

- `FP_OPTION_OPENSTRATEGY` — The approach used by `FPPool_Open()` to open connections to addresses in the connection string. Choices are:

  – `FP_NORMAL_OPEN` — `FPPool_Open()` attempts to open connections to all addresses in the connection string, and to all associated replication addresses. Consider using this strategy if your application performs numerous operations while the pool is open. This strategy is the default. This option is equivalent to `FP_OPTION_DEFAULT_OPTIONS`.

  – `FP_LAZY_OPEN` — `FPPool_Open()` opens connections to addresses only as needed. Consider using this strategy if your application frequently opens and closes the pool.

  Refer to *FPPool_Close* on page 1-12 and the *Centera Programmer's Guide*, P/N 069001127, for more information.

- `FP_OPTION_PROBE_LIMIT` — The threshold for how long an application probe is allowed to attempt communication with a node with the access role. The maximum threshold is 3600 seconds (1 hour). If a probe exceeds the limit, the SDK returns an error.

- FP_OPTION_RETRYCOUNT — The number of times an operation will be retried before a failure is reported to the client application. The default value is 6. If the first execution of the function fails, the system retries the function 6 times. In total the function executes 7 times. The maximum value is 99.

  If you do not want functions to retry automatically, set the retry count to 0.

  Note: Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on the retry mechanism.

- FP_OPTION_RETRYSLEEP — The time to wait before the failed API function call should be retried, in milliseconds. The maximum value is 100000 ms. If no retrysleep has been defined, the SDK uses an exponential back-off scheme. The sleep time increases after each retry, starting at 1 second, and doubles after each retry.
- FP_OPTION_MULTICLUSTER_*READ/WRITE/DELETE/EXISTS/QUERY*_STRATEGY — The operational failover behavior for each Centera operation. Table 1-6 lists the options.

  You can also control which cluster types can participate in the multicluster strategy. Refer to the descriptions of the FP_OPTION_MULTICLUSTER_*READ/WRITE/DELETE/EXISTS/QUERY*_CLUSTERS options.

You can also disable all operational failover for a given pool with `FP_OPTION_ENABLE_MULTICLUSTER_FAILOVER`. Refer to *FPPool_GetRetentionClassContext* on page 1-30.

Table 1-6 Multicluster Failover Strategy Options and Values

Option: Value	Description
FP_OPTION_MULTICLUSTER_READ_STRATEGY:	The multicluster failover strategy for read operations: FPClip_Open(), FPTag_BlobRead(), FPTag_BlobReadPartial().
FP_NO_STRATEGY	Content is read from the primary cluster only. If a connection to a node with the access role fails while a read is in progress, the next eligible node with the access role is used (an operation retry, not failover). Nodes with the access role from other clusters are not used. If connections to all nodes with the access role of the primary cluster fail, then the operation fails.
FP_FAILOVER_STRATEGY	If a connection to an node with the access role fails while a read is in progress, the next eligible node with the access role is used (an operation retry, not failover). If connections to all nodes with the access role of a cluster fail, then the next eligible cluster is used.
FP_REPLICATION_STRATEGY (Default)	If a connection to a node with the access role fails while a read is in progress, the next eligible node with the access role is used (an operation retry, not failover). If connections to all nodes with the access role of a cluster fail, then the next eligible cluster is used. In addition, if the content is not found on a given cluster, the read operation is replicated—performed on the next eligible cluster—which is not considered a retry.
FP_OPTION_MULTICLUSTER_WRITE_STRATEGY:	The multicluster failover strategy for write operations: FPClip_Write(), FPTag_BlobWrite().
FP_NO_STRATEGY (Default)	Content is written to the primary cluster only. If a connection to a node with the access role fails while a write is in progress and the stream can be reset, the next eligible node with the access role is used (an operation retry, not failover). If the stream cannot be reset, then the operation fails. Nodes with the access role from other clusters are not used. If connections to all nodes with the access role of a primary cluster fail, then the operation fails.
FP_FAILOVER_STRATEGY	Not supported.
FP_REPLICATION_STRATEGY	Content is written to all eligible clusters synchronously before the operation completes and the Content Address is returned. Note: • This strategy impacts performance. • This strategy can produce orphan content; that is, cases where the content is not written to all eligible clusters. In this case, the error information contains the Content Address and the cluster IDs where the content was written.

Table 1-6 Multicluster Failover Strategy Options and Values (continued)

Option: Value	Description
FP_OPTION_MULTICLUSTER_DELETE_STRATEGY:	The multicluster failover strategy for delete operations: FPClip_Delete(), FPClip_AuditedDelete(). This option also controls failover for the deprecated purge operations: FPClip_Purge(), FPTag_BlobPurge().
FP_NO_STRATEGY (Default)	Content is deleted from the primary cluster only. If a connection to a node with the access role fails while a delete is in progress, the next eligible node with the access role is used (an operation retry, not failover). Nodes with the access role from other clusters are not used. If connections to all nodes with the access role of a primary cluster fail, then the operation fails.
FP_FAILOVER_STRATEGY	Not supported.
FP_REPLICATION_STRATEGY	Content is deleted from all eligible clusters synchronously before the call returns.
FP_OPTION_MULTICLUSTER_EXISTS_STRATEGY:	The multicluster failover strategy for exists operations: FPClip_Exists(), FPTag_BlobExists().
FP_FAILOVER_STRATEGY (Default)	If a connection to a node with the access role fails while a check is in progress, the next eligible node with the access role is used (an operation retry, not failover). If connections to all nodes with the access role of a cluster fail, then the next eligible cluster is used.
FP_NO_STRATEGY	The existence of content is validated on the primary cluster only. If a connection to a node with the access role fails while a check is in progress, the next eligible node with the access role is used (an operation retry, not failover). nodes with the access role from other clusters are not used. If connections to all nodes with the access role of a primary cluster fail, then the operation fails.
FP_REPLICATION_STRATEGY	If a connection to a node with the access role fails while a check is in progress, the next eligible node with the access role is used (an operation retry, not failover). If connections to all nodes with the access role of a cluster fail, then the next eligible cluster is used. In addition, if the content is not found on a given cluster, the exists operation is replicated—performed on the next eligible cluster—which is not considered a retry. The resulting behavior is that the existence check returns true if the content is found on any of the eligible clusters.
FP_OPTION_MULTICLUSTER_QUERY_STRATEGY:	The multicluster failover strategy for query operations: FPPoolQuery_Open().

Table 1-6 Multicluster Failover Strategy Options and Values (continued)

Option: Value	Description
FP_NO_STRATEGY	Content is queried from the primary cluster only. If a connection to a node with the access role fails when starting a query, the next eligible node with the access role is used (an operation retry, not failover). Nodes with the access role from other clusters are not used. If connections to all nodes with the access role of a primary cluster fail, then the operation fails.
FP_FAILOVER_STRATEGY (Default)	If a connection to a node with the access role fails while starting a query, the next eligible node with the access role is used (an operation retry, not failover). If connections to all nodes with the access role of a cluster fail, then the next eligible cluster is used.
FP_REPLICATION_STRATEGY	A query is started on all eligible clusters. Query results are collated to preserve the increasing time sequence of results. Times are normalized to the primary cluster.

- `FP_OPTION_MULTICLUSTER_`*`READ/WRITE/DELETE/EXISTS/QUERY`*`_CLUSTERS`— The cluster types that are eligible for multicluster failover. In addition to defining the multicluster failover strategy for each SDK operation (refer to Table 1-6), you can specify which cluster types participate in that strategy for each operation. Choices are:
  - `FP_ALL_CLUSTERS`: The multicluster strategy uses all clusters in the pool. The typical order of access is: primary cluster, replica of the primary cluster, secondary clusters, and replicas of secondary clusters. The SDK, however, does not guarantee this order of access. This value is the default for read (`FP_OPTION_MULTICLUSTER_READ_CLUSTERS`), exists (`FP_OPTION_MULTICLUSTER_EXISTS_CLUSTERS`), and query (`FP_OPTION_MULTICLUSTER_QUERY_CLUSTERS`) operations.
  - `FP_PRIMARY_AND_PRIMARY_REPLICA_CLUSTER_ONLY`: The multicluster strategy uses the primary cluster and its replicas only. Secondary clusters and replicas of secondary clusters are not used.
  - `FP_NO_REPLICA_CLUSTERS`: The multicluster strategy uses secondary clusters, but no replica clusters.
  - `FP_PRIMARY_ONLY`: Operations are performed on the primary cluster only, irrespective of the multicluster strategy. This value is the default for write

(FP_OPTION_MULTICLUSTER_WRITE_CLUSTERS) and delete (FP_OPTION_MULTICLUSTER_DELETE_CLUSTERS) operations.

You identify clusters as being eligible to be primary clusters or not when you open the pool by specifying primary= and secondary= address prefixes in the connection string. Refer to *FPPool_Close* on page 1-12 for more information.

Note: These failover options are global to the application. Once set, the options should not be changed for different threads.

Note: You can disable failover for all capabilities for a given pool by setting FP_OPTION_ENABLE_MULTICLUSTER_FAILOVER to false. Refer to *FPPool_SetClipID* on page 1-35.

- const FPInt inOptionValue
  inOptionValue is the value for the given option.

Example:

```
RetryCount = 5;
FPPool_SetGlobalOption (FP_OPTION_RETRYCOUNT,
   RetryCount);

FPPool_SetGlobalOption (FP_OPTION_OPENSTRATEGY,
   FP_LAZY_OPEN);
```

Error Handling:

FPPool_GetLastError() returns ENOERR if successful. If unsuccessful, the following is a partial list of possible errors:

- FP_PARAM_ERR (program logic error)
- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_OUT_OF_BOUNDS_ERR (program logic error)
- FP_UNKNOWN_OPTION (program logic error)

FPPool_SetIntOption

Syntax:

```
FPPool_SetIntOption (const FPPoolRef inPool, const char
   *inOptionName, const FPInt inOptionValue)
```

Return Value:

```
void
```

Input Parameters:

```
const FPPoolRef inPool, const char *inOptionName, const
FPInt inOptionValue
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function sets the options for the given pool. To change global pool settings, refer to *FPPool_SetGlobalOption* on page 1-38.

Use the `FPPool_Open()` function to open and set the options for that pool.

Note: You can set global pool options as environment variables. The option settings made by an application take precedence and override those that were previously set as environment variables.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const char *inOptionName`
  `inOptionName` is a string with the name of the option to be set. The following initial pool options can be set:
  - `FP_OPTION_BUFFERSIZE` — The size of an internal C-Clip buffer in bytes. The default value is 16*1024. The minimum value is 1 KB. The maximum value is 10 MB.
  - `FP_OPTION_TIMEOUT` — The TCP/IP connection timeout in milliseconds. The default value is 120000 ms (2 minutes). The maximum value is 600000 ms (10 minutes).
  - `FP_OPTION_ENABLE_MULTICLUSTER_FAILOVER` — When this option is true (the default), multicluster failover is enabled. You can define the failover behavior for each capability using

FPPool_SetGlobalOption() (refer to page 1-38). By default this option is true (1). To turn multicluster failover off for all capabilities, specify false (0).

- FP_OPTION_DEFAULT_COLLISION_AVOIDANCE — This option can either be true (1) or false (0). This option is false by default. To enable collision avoidance at pool level set this option to true. If you enable this option, the SDK uses an additional blob discriminator for read and write operations of C-Clips and blobs. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on collision avoidance. To disable this option at pool level, reset the option to false. Collision avoidance can also be enabled or disabled at blob level, refer to *FPTag_BlobExists* on page 1-158.
- FP_OPTION_PREFETCH_SIZE — The size of the prefetch buffer. This buffer is used to assist in determining the size of the blob. The default size is 32 KB. The maximum size is 1 MB.

◆ `const FPInt inOptionValue`
`inOptionValue` is the value for the given option.

Example:

```
BufferSize = 32*1024;
FPPool_SetIntOption (myPool, FP_OPTION_BUFFERSIZE,
   BufferSize);
FPPool_SetIntOption (myPool,
   FP_OPTION_DEFAULT_COLLISION_AVOIDANCE, true);
```

Error Handling:

FPPool_GetLastError() returns ENOERR if successful. If unsuccessful, the following is a partial list of possible errors:

- ◆ FP_PARAM_ERR (program logic error)
- ◆ FP_WRONG_REFERENCE_ERR (program logic error)
- ◆ FP_OUT_OF_BOUNDS_ERR (program logic error)
- ◆ FP_UNKNOWN_OPTION (program logic error)

Clip Functions

The clip functions are a set of function calls that operate on C-Clips. A clip function can manipulate an entire C-Clip, retrieve information about a C-Clip, manipulate clip attributes, or manipulate a single tag from a C-Clip. Therefore the four groups of clip functions are:

- Clip handling
- Clip info
- Clip attribute
- Clip tag

The C-Clip must be open before you can perform a clip function (do not forget to close the C-Clip when finished).

C-Clips can be shared by multiple threads. Several threads can perform blob and tag operations within a C-Clip simultaneously.

Clip Handling Functions

This section describes the following functions that manipulate a C-Clip or C-Clip ID:

- `FPClip_AuditedDelete`
- `FPClip_Close`
- `FPClip_Create`
- `FPClip_Delete`
- `FPClip_EnableEBRWithClass`
- `FPClip_EnableEBRWithPeriod`
- `FPClip_Open`
- `FPClip_RawOpen`
- `FPClip_RawRead`
- `FPClip_RemoveRetentionClass`
- `FPClip_SetName`
- `FPClip_SetRetentionClass`
- `FPClip_SetRetentionHold`
- `FPClip_SetRetentionPeriod`
- `FPClip_TriggerEBREvent`
- `FPClip_TriggerEBREventWithClass`
- `FPClip_TriggerEBREventWithPeriod`
- `FPClip_Write`
- `FPClipID_GetCanonicalFormat`
- `FPClipID_GetStringFormat`

FPClip_AuditedDelete

Syntax:

```
FPClip_AuditedDelete (const FPPoolRef inPool, const
   FPClipID inClipID, const char *inReason, const FPLong
   inOptions)
```

Return Value:

```
void
```

Input Parameters:

```
const FPPoolRef inPool, const FPClipID inClipID, const
char *inReason, const FPLong inOptions)
```

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function deletes the given CDF from the first writable cluster of a given pool and records audit information describing the reason for the deletion.

The delete operation succeeds only if:

- The "delete" capability is enabled, or in the case of a privileged deletion, the "privileged-delete" capability is enabled. Refer to *FPPool_GetCapability* on page 1-13 for information on the server capabilities. The function returns `FP_OPERATION_NOT_ALLOWED` if the required profile capability is false.

Note: A Compliance Edition Plus model never allows a privileged deletion.

- The retention period of the C-Clip has expired, or you have requested and the profile capability allows privileged deletion. Refer to *FPClip_SetRetentionPeriod* on page 1-72 and *FPClip_SetRetentionClass* on page 1-68 for information on retention periods. The function returns `FP_OPERATION_NOT_ALLOWED` if the retention period has not expired, or you requested but do not have permission for a privileged deletion.
- All copies of the CDF have been successfully removed.

Delete operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N

069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Note that this function does not delete associated blobs. Garbage collection automatically deletes blobs that are no longer referenced by any C-Clip. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on deleting data.

Deleting a C-Clip leaves a reflection—metadata about the deleted C-Clip—on the cluster. Reflections are only exposed in the SDK through the query functions. Refer to *Query Functions* on page 1-223. The audit string, if specified, is recorded in the reflection.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const FPClipID inClipID`
  The ID of a C-Clip.
- `const char *inReason`
  The reason for the delete operation, which is recorded as an audit string in the reflection. Specify `NULL` if you do not want to record an audit string. An audit string is required for privileged deletions. The string must be smaller than 16 KB.
- `const FPLong inOptions`
  Specify one of the following options:
  - `FP_OPTION_DEFAULT_OPTIONS` — Specify this option if you are not performing a privileged deletion.
  - `FP_OPTION_DELETE_PRIVILEGED` — Delete the C-Clip even if the retention period has not expired. You must specify an `inReason` string when performing a privileged deletion. Note that a Compliance Edition Plus model never allows a privileged deletion.

Example:

```
FPClip_AuditedDelete (myPool, myClipID, "Employee has
   left the company.", FP_OPTION_DELETE_PRIVILEGED);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_PROBEPACKET_ERR` (internal error)

- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_SERVER_ERR (server error)
- FP_CONTROLFIELD_ERR (server error)
- FP_NUMLOC_FIELD_ERR (server error)
- FP_UNKNOWN_OPTION (internal error)
- FP_CLIP_NOT_FOUND_ERR (program logic error)
- FP_VERSION_ERR (internal error)
- FP_NOT_RECEIVE_REPLY_ERR (network error)
- FP_SEGDATA_ERR (internal error)
- FP_SERVER_NOTREADY_ERR (server error)
- FP_POOLCLOSED_ERR (program logic error)
- FP_BLOBBUSY_ERR (server error)
- FP_OPERATION_NOT_ALLOWED (client error)
- FP_TRANSACTION_FAILED_ERR (server error)
- Additional server errors

FPClip_Close

Syntax: `FPClip_Close (const FPClipRef inClip)`

Return Value: `void`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference.

Description: This function closes the given C-Clip and frees up all memory allocated to the C-Clip. Note that calling this function on a C-Clip that has already been closed may produce unwanted results.

Note: If you close a C-Clip before calling `FPClip_Write()`, then any modifications to the C-Clip will be lost.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
FPClip_Close (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OBJECTINUSE_ERR` (client error)

FPClip_Create

Syntax:

```
FPClip_Create (const FPPoolRef inPool, const char
   *inName)
```

Return Value:

```
FPClipRef
```

Input Parameters:

```
const FPPoolRef inPool, const char *inName
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function creates a new, empty C-Clip in memory. This function returns a reference to the in-memory C-Clip.

During the execution of `FPClip_Create()`, the SDK looks for the environment variable `CENTERA_CUSTOM_METADATA`. This variable may contain a comma-separated list of environment variables that will be added to the CDF during `FPClip_Write()`. The number of metadata items is limited by memory (100 MB). The metadata can be retrieved using `FPClip_GetDescriptionAttribute()`. No error is reported if the function cannot access a metadata item.

For example, if the `CENTERA_CUSTOM_METADATA` variable is defined as:

```
CENTERA_CUSTOM_METADATA=USER,APPLICATION,HOSTNAME
```

and the referenced environment variables are defined as:

```
USER=Doe
APPLICATION=RWE Exerciser
HOSTNAME=QA Test 15
```

then the SDK adds the following information to the CDF:

```
<custom-meta name="USER" value="Doe">
<custom-meta name="APPLICATION" value="RWE Exerciser">
<custom-meta name="HOSTNAME" value="QA Test 15">
```

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const char *inName`
  `inName` is a string holding the name of the C-Clip. If `inName` is `NULL`, the name of the C-Clip is `untitled`.

Example:

```
myClip = FPClip_Create (myPool, "anotherclip");
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_TAGTREE_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_TAG_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_Delete

Syntax:

```
FPClip_Delete (const FPPoolRef inPool, const FPClipID
   inClipID)
```

Return Value:

```
void
```

Input Parameters:

```
const FPPoolRef inPool, const FPClipID inClipID
```

Concurrency Requirement:

This function is thread safe.

Description:

This function deletes the CDF for the specified Clip ID from the first writable cluster of a given pool if the retention period of the C-Clip has expired and if the server capability "delete" is enabled. Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities, and to *FPClip_SetRetentionPeriod* on page 1-72 and *FPClip_SetRetentionClass* on page 1-68 for more information on retention periods.

Delete operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

To specify a reason for the deletion (an audit string) or to delete a C-Clip before the retention period has expired, refer to *FPClip_AuditedDelete* on page 1-48.

A C-Clip will only be deleted when all copies of the CDF have been successfully removed.

This function returns `FP_OPERATION_NOT_ALLOWED` if the retention period has not yet expired or if the "delete" capability is disabled. In that case, the CDF is not deleted.

Note that this function does not itself delete associated blobs. Garbage collection automatically deletes blobs that are no longer referenced by any C-Clip. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on deleting data.

Note: The server allows the application to perform this call if the server capability "delete" is enabled. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Deleting a C-Clip leaves a reflection—metadata about the deleted C-Clip—on the cluster. Reflections are only exposed in the SDK through the query functions. Refer to *Query Functions* on page 1-223.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const FPClipID inClipID`
  The ID of a C-Clip.

Example:

```
FPClip_Delete (myPool, myClipID);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NUMLOC_FIELD_ERR` (server error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_CLIP_NOT_FOUND_ERR` (program logic error)
- `FP_VERSION_ERR` (internal error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_TRANSACTION_FAILED_ERR` (server error)

FPClip_EnableEBRWithClass

Syntax:

```
FPClip_EnableEBRWithClass (const FPClipRef inClip,
   const FPRetentionClassRef inClass)
```

Return Value:

`void`

Input Parameters:

`const FPClipRef inClip, const FPRetentionClassRef inClass`

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function sets a C-Clip to be eligible to receive a future event and enables an event-based retention (EBR) class to be assigned to the C-Clip during C-Clip creation time.

The retention period associated with the event-based retention class is added to the server time at the time of the triggering of the event to determine whether the C-Clip can be deleted. For more information, refer to *FPClip_TriggerEBREventWithClass* on page 1-76 or *FPClip_TriggerEBREvent* on page 1-74.

When `FPClip_EnableEBRWithClass()` is called, you must specify an existing retention class.

When writing a C-Clip and enabling EBR, you can also specify a fixed retention period using `FPClip_SetRetention Class/Period()`. If a fixed retention period is specified, it is subject to the fixed retention minimum/maximum rule. However, if the fixed retention period does not exist for the C-Clip, the SDK sets it to zero (`0`), instead of to the default retention period. In this case, it is not subject to the fixed retention minimum/maximum rule.

Note: As enabling a C-Clip for EBR modifies the C-Clip, you must enable EBR prior to writing the C-Clip to Centera. Writing a modified C-Clip creates a new C-Clip with a new clip ID, and does not change the existing C-Clip.

Notes: A C-Clip with event-based retention enabled cannot be deleted until after the triggering event is received and all the associated retention periods have expired. In this case, the triggering event is the call to `FPClip_TriggerEBREvent()`, `FPClip_TriggerEBREvent WithClass()`, or to `FPClip_TriggerEBREventWithPeriod()`.

The only circumstance where an EBR-enabled C-Clip can be deleted is with a Privileged-Delete operation (for CE mode only), even if the event has yet to trigger or any other associated retention period has yet to expire.

When `FPClip_EnableEBRWithClass()` is called, the SDK refers to the `FP_COMPLIANCE` capability to verify that the application holds a license for Advanced Retention Management. If unsupported, the SDK immediately generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`. (Refer to *FPPool_GetCapability* on page 1-13.)

Note: For more information on event-based retention, refer to Chapter 5, *Data Retention,* in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const FPRetentionClassRef inClass`
  The reference to the retention class used to indirectly set the wait period of the C-Clip.

Example:

```
FPRetentionClassContextRef vContextRef;
FPRetentionClassRef vClassRef;
vContextRef = FPPool_GetRetentionClassContext (inPool);
vClassRef = FPRetentionClassContext_GetFirstClass
   (vContextRef);
FPClip_EnableEBREventWithClass (myClip, vClassRef);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_EBR_OVERRIDE_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_EnableEBRWithPeriod

Syntax:

```
FPClip_EnableEBRWithPeriod (const FPClipRef inClip,
   FPLong inSeconds)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, FPLong inSeconds
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function sets a C-Clip to be eligible to receive a future event and enables an event-based retention (EBR) period to be assigned to the C-Clip during C-Clip creation time.

This EBR retention period is added to the server time at the time of the triggering of the event to determine whether the C-Clip can be deleted. For more information, refer to *FPClip_TriggerEBREventWithPeriod* on page 1-78, *FPClip_TriggerEBREventWithClass* on page 1-76, and *FPClip_TriggerEBREvent* on page 1-74.

The value of the EBR retention period must be greater than or equaled to zero (`0`), except where the default value of `FP_INFINITE_RETENTION_PERIOD` or `FP_DEFAULT_RETENTION_PERIOD` is applied. This value is subject to the minimum/maximum rule for event-based retention periods.

If you use `FP_DEFAULT_RETENTION_PERIOD` (`-2`) to specify the EBR period in `FPClip_EnableEBRWithPeriod()`, note that the corresponding default value on the server is subject to the minimum/maximum rule for event-based retention. If the default value is not within the range, the SDK returns the error `FP_RETENTION_OUT_OF_BOUNDS_ERR`.

EMC recommends that you specify at least a minimum event-based retention period instead of using `FP_NO_RETENTION_PERIOD` (`0`) or `FP_DEFAULT_RETENTION_PERIOD` (`-2`) to specify the EBR period.

- Specifying `FP_NO_RETENTION_PERIOD` allows a C-Clip to be deleted immediately after triggering the event.
- Specifying `FP_DEFAULT_RETENTION_PERIOD` allows the SDK to use the default retention period, which results in C-Clips that never can be deleted on a CE+ cluster.

When writing a C-Clip and enabling EBR, you can also specify a fixed retention period using `FPClip_SetRetentionPeriod()`. If a fixed retention period is specified, it is subject to the fixed retention minimum/maximum rule. However, if the fixed retention period does not exist for the C-Clip, the SDK sets it to zero (`0`), instead of to the default retention period. In this case, it is not subject to the fixed retention, minimum/maximum rule.

Note: As enabling a C-Clip for EBR modifies the C-Clip, you must enable EBR prior to writing the C-Clip to Centera. Writing a modified C-Clip creates a new C-Clip with a new clip ID, and does not change the existing C-Clip.

Notes: A C-Clip with event-based retention enabled cannot be deleted until after the triggering event is received and all the associated retention periods have expired. In this case, the triggering event is the call to `FPClip_TriggerEBREvent()`, `FPClip_TriggerEBREvent WithPeriod()`, or to `FPClip_TriggerEBREventWithClass()`.

The only circumstance where an EBR-enabled C-Clip can be deleted is with a Privileged-Delete operation (for CE mode only), even if the event has yet to trigger or any other associated retention period has yet to expire.

When `FPClip_EnableEBRWithPeriod()` is called, the SDK refers to the `FP_COMPLIANCE` capability to verify that the application holds a license for Advanced Retention Management. If unsupported, the SDK immediately generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`. (Refer to *FPPool_GetCapability* on page 1-13.)

Note: For more information on event-based retention, refer to Chapter 5, *Data Retention,* in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `FPLong inSeconds`
  The retention period (in seconds) that is to be measured from when the triggering event occurs.

Example:

```
FPClip_EnableEBRWithPeriod (myClip, 400);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_EBR_OVERRIDE_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_RETENTION_OUT_OF_BOUNDS_ERR` (server error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_Open

Syntax:

```
FPClip_Open (const FPPoolRef inPool, const FPClipID
   inClipID, const FPInt inOpenMode)
```

Return Value:

```
FPClipRef
```

Input Parameters:

```
const FPPoolRef inPool, const FPClipID inClipID, const
FPInt inOpenMode
```

Concurrency Requirement:

This function is thread safe.

Description:

With `FPClip_Open()`, you can open a stored C-Clip as a tree structure or as a flat structure. This function reads the CDF into the memory of the application server and returns a reference to the opened C-Clip.

The server allows the application to perform this call if the server capability "read" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned. Refer to *FPPool_GetCapability* on page 1-13 for more information on server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Read operations fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Note: This function keeps the C-Clip data in a memory buffer of which the size has been specified with `FPPool_SetIntOption(buffersize)`. Any overflow is temporarily stored on disk.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const FPClipID inClipID`
  The C-Clip ID returned by `FPClip_Write`.()
- `const FPInt inOpenMode`
  The method of opening a C-Clip. Choices are:

- FP_OPEN_ASTREE — Opens the C-Clip as a tree structure in read/write mode and enables hierarchical navigation through the C-Clip tags. Refer to *FPClip_GetTopTag* on page 1-121 for more information.
- FP_OPEN_FLAT — Opens the C-Clip as a flat structure in read-only mode and enables sequential access within the C-Clip. This option is optimal for opening C-Clips that do not fit in memory. Refer to *FPClip_GetTopTag* on page 1-121 for more information.

Example:

```
myClip = FPClip_Open (myPool, myClipID, FP_OPEN_ASTREE);
```

Error Handling: FPPool_GetLastError() returns ENOERR if successful. If unsuccessful, the following is a partial list of possible errors:

- FP_UNKNOWN_OPTION (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- FP_PARAM_ERR (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- FP_CLIP_NOT_FOUND_ERR (program logic error)
- FP_VERSION_ERR (internal error)
- FP_WRONG_REFERENCE_ERR (program logic error)
- FP_SECTION_NOT_FOUND_ERR (internal error)
- FP_PROBEPACKET_ERR (internal error)
- FP_NO_POOL_ERR (network error)
- FP_NO_SOCKET_AVAIL_ERR (network error)
- FP_PROTOCOL_ERR (internal error)
- FP_SERVER_ERR (server error)
- FP_CONTROLFIELD_ERR (server error)
- FP_NOT_RECEIVE_REPLY_ERR (network error)
- FP_SEGDATA_ERR (internal error)
- FP_BLOBIDMISMATCH_ERR (server error)
- FP_SERVER_NOTREADY_ERR (server error)
- FP_TAG_NOT_FOUND_ERR (internal error)
- FP_BLOBBUSY_ERR (server error)
- FP_OPERATION_NOT_ALLOWED (client error)

FPClip_RawOpen

Syntax:

```
FPClip_RawOpen (const FPPoolRef inPool, const FPClipID
   inClipID, const FPStreamRef inStream, const FPLong
   inOptions)
```

Return Value:

`FPClipRef`

Input Parameters:

`const FPPoolRef inPool, const FPClipID inClipID, const FPStreamRef inStream, const FPLong inOptions`

Concurrency Requirement:

This function is thread safe.

Description:

This function reads the content of `inStream` and creates a new in-memory C-Clip. The new C-Clip ID must match the given C-Clip ID.

If the Storage Strategy Performance scheme is used to create the new C-Clip, this function returns FP_OPERATION_NOT_ALLOWED when used against C-Clips from a Centera prior to version 2.1.

When the C-Clip has been created in memory, the function returns a reference to that C-Clip. This function returns `NULL` if no C-Clip has been built.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const FPClipID inClipID`
  The C-Clip used to reference the C-Clip that has to be read from the stream.
- `const FPStreamRef inStream`
  The reference to an input stream. Marking support is not necessary.
- `const FPLong inOptions`
  Reserved for future use. Specify `FP_OPTION_DEFAULT_OPTIONS`.

Example:

```
NewClip = FPClip_RawOpen (myPool, myClipID, myStream,
   FP_OPTION_DEFAULT_OPTIONS);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_STREAM_ERR` (client error)
- `FP_VERSION_ERR` (internal error)
- stream related errors (refer to *Stream Functions* on page 1-189 for more information)

FPClip_RawRead

Syntax:

```
FPClip_RawRead (const FPClipRef inClip, const FPStreamRef
   inStream)
```

Return value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const FPStreamRef inStream
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference.

Description:

This function reads the content of the CDF into `inStream`. If the C-Clip has been modified, that is if `FPClip_IsModified()` returns true, the function rewrites the tag tree into `inStream`.

By using this function, the application can store the stream content on another device for subsequent restore operations of the C-Clip. The application must not change the stream content as it is the source for the input stream of `FPClip_RawOpen()`.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip returned by `FPClip_Open()`.
- `const FPStreamRef inStream`
  The reference to a stream that has been created by an `FPStream_CreateXXX()` function or a generic stream for writing. The stream is not required to support marking.

Example:

```
FPClip_RawRead (myClip, myStream);
```

Error Handling:

- `FP_PARAM_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_STREAM_ERR` (client error)
- stream related errors (refer to *Stream Functions* on page 1-189 for more information)

FPClip_RemoveRetentionClass

Syntax:

```
FPClip_RemoveRetentionClass (const FPClipRef inClipRef)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClipRef,
const FPRetentionClassRef inClassRef
```

Concurrency Requirement:

This function is thread safe.

Description:

This function removes a previously associated retention class from the specified in-memory C-Clip. Refer to *FPClip_SetRetentionClass* on page 1-68 for more information.

Parameters:

`const FPClipRef inClipRef`
The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
FPClip_RemoveRetentionClass (vClip);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)

FPClip_SetName

Syntax:

```
FPClip_SetName (const FPClipRef inClip, const char
   *inClipName)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const char *inClipName
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function changes the name of the given C-Clip to the name given in `inClipName`.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `const char *inClipName`
  `inClipName` is a string holding the new name of the C-Clip.

Example:

```
FPClip_SetName (myClip, "newclipname");
```

Error Handling:

`FPPool_GetLastError()` returns ENOERR if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_SetRetentionClass

Syntax:

```
FPClip_SetRetentionClass (const FPClipRef inClipRef,
   const FPRetentionClassRef inClassRef)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClipRef,
const FPRetentionClassRef inClassRef
```

Concurrency Requirement:

This function is thread safe.

Description:

This function sets the retention class of the given C-Clip. The retention class defines the retention period for the C-Clip. A retention period specifies how long a C-Clip has to be stored before it can be deleted.

Using retention classes is an alternative to assigning an integral retention period to a C-Clip. Refer to *FPClip_SetRetentionPeriod* on page 1-72 for more information.

If a C-Clip does not have an explicitly assigned retention period or class, the C-Clip will be stored with the retention period that is specified by the cluster (refer to *FPPool_GetCapability* on page 1-13 for the possible cluster settings).

To remove a retention class association from an in-memory C-Clip, call `FPClip_RemoveRetentionClass()`.

Calling this function clears any existing retention period associated with the C-Clip.

For more information on retention periods and classes, refer to the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClipRef`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `FPRetentionClassRef inClassRef`
  The reference to a retention class as returned by one of the `FPRetentionClassContext_Get`*`XXX`*`()` functions.

Example:

```
FPClip_SetRetentionClass (vClip, vRetClass);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_OUT_OF_MEMORY_ERR` (client error)

FPClip_SetRetentionHold

Syntax:

```
FPClip_SetRetentionHold (const FPClipRef inClip,
   const FPBool inHoldFlag, const char* inHoldID)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const FPBool inHoldFlag,
const char* inHoldID
```

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function sets a retention hold on a C-Clip, which effectively prevents the C-Clip from being deleted in Centera. This setting supersedes the expiration of any other retention policy that may be in effect. In addition, a Privileged-Delete operation on a Governance Edition (CE mode) can never be performed on a C-Clip that is under retention hold.

Changing the hold state of a C-Clip does not affect the content address of a C-Clip, as long as no other changes to the C-Clip are made, which would create a new C-Clip. In this case, the result of the call puts the hold on the new C-Clip and not on the original one.

A C-Clip can have multiple retention hold names assigned to it—up to a maximum of 100. If this is the case, each hold name requires a separate API call with a unique identifier for the hold. Each hold ID can have up to a maximum of 64 characters.

Note: The application must generate the unique hold IDs and be able to track the specific holds associated with a C-Clip. You cannot query a C-Clip for this information.

When `FPClip_SetRetentionHold()` is called, the SDK refers to the `FP_COMPLIANCE` capability to verify that the application holds a license for Advanced Retention Management. If unsupported, the SDK immediately generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`. In addition, the SDK refers to the `FP_RETENTION_HOLD` capability to verify that retention hold is allowed. If not allowed, the SDK generates the error

FP_OPERATION_NOT_ALLOWED. (Refer to *FPPool_GetCapability* on page 1-13.)

Note: For more information on retention hold and event-based retention (EBR), refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const FPBool inHoldFlag`
  The reference to the Boolean value that indicates whether the hold state of the C-Clip is enabled or disabled.

  Note: If `inHoldFlag` is set to `False`, it removes the specified retention hold from the C-Clip.
- `const char* inHoldID`
  The reference to the ID that is the name of the hold. The hold ID may contain up to 64 characters. You can assign multiple holds to a single C-Clip, up to 100 in total.

Example:

```
FPClip_SetRetentionHold (myClip, true, "myHold1");
FPClip_SetRetentionHold (myClip, true, "myHold2");
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (program logic error)
- `FP_INVALID_NAME` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_OUT_OF_MEMORY_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_RETENTION_HOLD_COUNT_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_SetRetentionPeriod

Syntax:

```
FPClip_SetRetentionPeriod (const FPClipRef inClip, const
   FPLong inRetentionSecs)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const FPLong inRetentionSecs
```

Concurrency Requirement:

This function is thread safe.

Description:

This function sets the specific retention period, in seconds, of the given C-Clip. A retention period specifies how long a C-Clip has to be stored before it can be deleted.

If you use `FP_DEFAULT_RETENTION_PERIOD` (`-2`) to specify the fixed period in `FPClip_SetRetentionPeriod()`, note that the corresponding default value on the server is subject to the minimum/maximum rule for fixed retention. If the default value is not within the range, the SDK returns the error `FP_RETENTION_OUT_OF_BOUNDS_ERR`.

An alternative to assigning an integral retention period to a C-Clip is to use retention classes. Refer to *FPClip_SetRetentionClass* on page 1-68 for more information.

If a C-Clip does not have an integral retention period or associated retention class, the C-Clip will be stored with the retention period that is associated with the cluster (refer to *FPPool_GetCapability* on page 1-13 for the possible cluster settings).

Calling this function clears any existing retention class associated with the C-Clip.

For more information on retention periods and classes, refer to the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const FPLong inRetentionSecs`
  The retention period, in seconds, or one of the following values:
  - `FP_NO_RETENTION_PERIOD` — The C-Clip can be deleted at any time.

- `FP_INFINITE_RETENTION_PERIOD` — The C-Clip can never be deleted, except possibly by a Privileged Delete operation (refer to *FPClip_AuditedDelete* on page 1-48).
- `FP_DEFAULT_RETENTION_PERIOD` — The retention period is based on the mode of the cluster that manages the C-Clip. For a Compliance Edition Plus model, the default retention period is infinite. For a Governance Edition model, the default is no retention period or as defined by the system administrator.

Example:

```
FPClip_SetRetentionPeriod(vClip,
   FP_INFINITE_RETENTION_PERIOD);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_TriggerEBREvent

Syntax: `FPClip_TriggerEBREvent (const FPClipRef inClip)`

Return Value: `void`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function triggers the event of a C-Clip for which event-based retention (EBR) was enabled, and effectively starts the run time of the associated EBR retention period that was previously set in `FPClip_EnableEBRWithClass()` or `FPClip_EnableEBRWithPeriod()`.

An EBR event can be triggered only once. If an attempt is made to trigger the event multiple times, the call returns the `FP_EBR_OVERRIDE_ERR` error.

When `FPClip_TriggerEBREvent()` is called, the SDK refers to the `FP_COMPLIANCE` capability to verify that the application holds a license for Advanced Retention Management. If unsupported, the SDK immediately generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`. (Refer to *FPPool_GetCapability* on page 1-13.)

Note: A C-Clip can be deleted after the triggering event is received, and both the fixed and event-based retention periods have expired. Privileged Delete calls override expiration rules for fixed and event-based retention.

Note: For more information about event-based retention, refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
FPClip_TriggerEBREvent (myClip);
```

Error Handling: `FPPool_GetLastError()` returns ENOERR if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_NON_EBR_CLIP_ERR` (server error)
- `FP_EBR_OVERRIDE_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_TriggerEBREventWithClass

Syntax:

```
FPClip_TriggerEBREventWithClass (const FPClipRef inClip,
   const FPRetentionClassRef inClass)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const FPRetentionClassRef inClass
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function triggers the event of a C-Clip for which event-based retention (EBR) was enabled, sets a new EBR period for the C-Clip, and effectively starts the run time of the applicable EBR retention period.

An EBR event can be triggered only once. If an attempt is made to trigger the event multiple times, the call returns the `FP_EBR_OVERRIDE_ERR` error.

Note: When triggering the EBR event, do not make any changes to the C-Clip. Doing so creates a new C-Clip, and the API call results in the event triggering on the new C-Clip and not on the original one.

If the period specified at trigger time is less than the period specified for the C-Clip when `FPClip_EnableEBRWithClass/Period()` was called, the server ignores the trigger period and applies the period specified at enable time. The server enforces whichever EBR retention period is the longer of the two periods.

When `FPClip_TriggerEBREventWithPeriod()` is called, the SDK refers to the `FP_COMPLIANCE` capability to verify that the application holds a license for Advanced Retention Management. If unsupported, the SDK immediately generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`. (Refer to *FPPool_GetCapability* on page 1-13.)

Note: A C-Clip for which the event is triggered cannot be deleted until the fixed retention and the longer of the two event-based retention periods (previously set with `FPClip_EnableEBRWithPeriod()` or `FPClip_EnableEBRWithClass()` and the one specified in this API call) have expired. Privileged Delete calls override expiration rules for fixed and event-based retention on a cluster in CE mode (Governance Edition).

Note: For more information about event-based retention, refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `const FPRetentionClassRef inClass`
  `inClass` is a reference to the retention class that indirectly sets and starts the run time of the retention period from the time this EBR event (API call) is triggered.

Example:

```
FPRetentionClassContextRef vContextRef;
FPRetentionClassRef vClassRef;
vContextRef = FPPool GetRetentionClassContext (inPool);
vClassRef = FPRetentionClassContext GetLastClass
   (vContextRef);
FPClip_TriggerEBREventWithClass (myClip, vClassRef);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_NON_EBR_CLIP_ERR` (server error)
- `FP_EBR_OVERRIDE_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_TriggerEBREventWithPeriod

Syntax:

```
FPClip_TriggerEBREventWithPeriod (FPClipRef inClip,
   FPLong inSeconds)
```

Return Value:

```
void
```

Input Parameters:

```
FPClipRef inClip, FP Long inSeconds
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function triggers the event of a C-Clip for which event-based retention (EBR) was enabled, sets a new EBR retention period for the C-Clip, and effectively starts the run time of the applicable EBR retention period.

An EBR event can be triggered only once. If an attempt is made to trigger the event multiple times, the call returns the `FP_EBR_OVERRIDE_ERR` error.

If the period specified at trigger time is less than the period specified for the C-Clip when `FPClip_EnableEBRWithClass/Period()` was called, the server ignores the trigger period and applies the period specified at enable time. The server enforces whichever EBR retention period is the longer of the two periods.

The value of the EBR retention period is subject to the minimum/maximum rule for event-based retention periods.

If you use `FP_DEFAULT_RETENTION_PERIOD` (`-2`) to specify the EBR period in `FPClip_TriggerEBREventWithPeriod()`, note that the corresponding default value on the server is subject to the minimum/maximum rule for event-based retention. If the default value is not within the range, the SDK returns the error `FP_RETENTION_OUT_OF_BOUNDS_ERR`.

EMC recommends that you specify at least a minimum event-based retention period instead of using `FP_NO_RETENTION_PERIOD` (`0`) or `FP_DEFAULT_RETENTION_PERIOD` (`-2`) to specify the EBR period.

- Specifying `FP_NO_RETENTION_PERIOD` allows a C-Clip to be deleted immediately after triggering the event.
- Specifying `FP_DEFAULT_RETENTION_PERIOD` allows the SDK to use the default retention period, which results in C-Clips that never can be deleted on a CE+ cluster.

Note: When triggering the EBR event, do not make any changes to the C-Clip. Doing so creates a new C-Clip, and the API call results in the event triggering on the new C-Clip and not on the original one.

When `FPClip_TriggerEBREventWithPeriod()` is called, the SDK refers to the `FP_COMPLIANCE` capability to verify that the application holds a license for Advanced Retention Management. If unsupported, the SDK immediately generates the error `FP_ADVANCED_RETENTION_DISABLED_ERR`. (Refer to *FPPool_GetCapability* on page 1-13.)

Note: A C-Clip for which the event is triggered cannot be deleted until the fixed retention and the longer of the two event-based retention periods (previously set with `FPClip_EnableEBRWithPeriod()` or `FPClip_EnableEBRWithClass()` and the one specified in this API call) have expired. Privileged Delete calls override expiration rules for fixed and event-based retention on a cluster in CE mode (Governance Edition).

Note: For more information about event-based retention, refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `FPLong inSeconds`
  `inSeconds` is a value in seconds that explicitly sets and starts the run time of the retention period from the time this EBR event (API call) is triggered.

Example:

```
FPClip_TriggerEBREventWithPeriod (myClip, 300);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_EBR_OVERRIDE_ERR` (server error)
- `FP_NON_EBR_CLIP_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_RETENTION_OUT_OF_BOUNDS_ERR` (server error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_Write

Syntax: `FPClip_Write (const FPClipRef inClip, FPClipID outClipID)`

Return Value: `void`

Input Parameters: `const FPClipRef inClip`

Output Parameters: `FPClipID outClipID`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function writes the content of a C-Clip to the pool as a CDF and returns the C-Clip ID (Content Address). This address is 64 bytes.

If collision avoidance is enabled at pool level, refer to *FPPool_SetClipID* on page 1-35, this function returns: <C-CLIPID><REFID>. For example: `42L0M726P04T2e7QU2445E81QBK7QU2445E81QBK42L0M726P04T2.` Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on collision avoidance.

If Storage Strategy Performance is enabled on the server, files smaller than the server-defined threshold (by default 250 KB) will have a Content Address similar to the one that is created when Collision Avoidance has been enabled.

Write operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

The server allows the application to perform this call if the server capability "write" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned. This error is also returned if the C-Clip has been opened in flat mode (read only).

Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Note: This function keeps the C-Clip data in a memory buffer of which the size has been specified with `FPPool_SetIntOption(buffersize)`. Any overflow is temporarily stored on disk.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `FPClipID outClipID`
  The C-Clip ID that the function returns.

Example:

```
FPClip_Write (myClip, myClipID);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ACK_NOT_RCV_ERR` (server error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_BLOBIDFIELD_ERR` (server error)
- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_DUPLICATE_FILE_ERR` (internal error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)
- `FP_PARAM_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_SERVER_NO_CAPACITY_ERR` (server error)
- `FP_STACK_DEPTH_ERR` (program logic error)
- `FP_TAGTREE_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClipID_GetCanonicalFormat

Syntax:
```
FPClipID_GetCanonicalFormat (const FPClipID inClipID,
   FPCanonicalClipID outClipID)
```

Return Value:
```
void
```

Input Parameters:
```
const FPClipID inClipID
```

Output Parameters:
```
FPCanonicalClipID outClipID
```

Concurrency Requirement: This function is thread safe.

Description: This function converts the string representation of a Content Address into the platform-neutral canonical format. The canonical format is ideal for storing Content Addresses in databases.

Parameters:

- `const FPClipID inClipID`
  The reference to a C-Clip ID that holds the string format of a Content Address.
- `FPCanonicalClipID outClipID`
  The return of the C-Clip ID of the Content Address stored in canonical format.

Example:
```
FPClipID vClipID;
FPClip_Write(vClip, vClipID);
FPCanonicalClipID vCanonicalClipID;
FPClipID_GetCanonicalFormat (vClipID, vCanonicalClipID);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_CLIP_NOT_FOUND_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClipID_GetStringFormat

Syntax:

```
FPClipID_GetStringFormat (const FPCanonicalClipID
   inClipID, FPClipID outClipID)
```

Return Value:

```
void
```

Input Parameters:

```
const FPCanonicalClipID inClipID
```

Output Parameters:

```
FPClipID outClipID
```

Concurrency Requirement:

This function is thread safe.

Description:

This function converts the canonical representation of a Content Address into the platform-specific string format. The string format of a Content Address is ideal for sharing in email or in other text-based media.

Parameters:

- `const FPCanonicalClipID inClipID`
  The reference to a C-Clip ID that holds the canonical format of a Content Address.
- `FPClipID outClipID`
  The return of the C-Clip ID of the Content Address stored in string format.

Example:

```
FPClipID vClipID;
FPClipID_GetStringFormat(vCanonicalClipID, vClipID);
FPClipRef vClip = FPClip_Open(vPool, vClipID,
   FP_OPEN_ASTREE);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of what errors can occur.

- `FP_CLIP_NOT_FOUND_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

Clip Info Functions

This section describes the following functions that retrieve information about a C-Clip:

- FPClip_Exists
- FPClip_GetClipID
- FPClip_GetCreationDate
- FPClip_GetEBRClassName
- FPClip_GetEBREventTime
- FPClip_GetEBRPeriod
- FPClip_GetName
- FPClip_GetNumBlobs
- FPClip_GetNumTags
- FPClip_GetPoolRef
- FPClip_GetRetentionClassName
- FPClip_GetRetentionHold
- FPClip_GetRetentionPeriod
- FPClip_GetTotalSize
- FPClip_IsEBREnabled
- FPClip_IsModified
- FPClip_ValidateRetentionClass

FPClip_Exists

Syntax:

```
FPClip_Exists (const FPPoolRef inPool, const FPClipID
   inClipID)
```

Return Value:

```
FPBool
```

Input Parameters:

```
const FPPoolRef inPool, const FPClipID inClipID
```

Concurrency Requirement:

This function is thread safe.

Description:

This function determines if the given C-Clip exists in the given pool and returns true or false.

Exists operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Note: The server allows the application to perform this call if the server capability "exist" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Parameters:

- `const FPPoolRef inPool`
  The reference to a pool opened by `FPPool_Open()`.
- `const FPClipID inClipID`
  The ID of a C-Clip.

Example:

```
FPClip_Exists (myPool, myClipID);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SERVER_ERR` (server error)
- `FP_NUMLOC_FIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_CLIP_NOT_FOUND_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)

FPClip_GetClipID

Syntax:

```
FPClip_GetClipID (const FPClipRef inClip, FPClipID
   outClipID)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip
```

Output Parameters:

```
FPClipID outClipID
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function retrieves the ID of the given C-Clip and returns it in `outClipID`.

This function returns an empty string for a C-Clip created by `FPClip_Create()` but that has not yet been written to the pool by `FPClip_Write()`.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `FPClipID outClipID`
  The C-Clip ID as specified in the `FPClip_Open()` function or as modified by the `FPClip_Write()` function (can be empty).

Example:

```
FPClip_GetClipID (myClip, myClipID);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_TAG_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_GetCreationDate

Syntax:

```
FPClip_GetCreationDate (const FPClipRef inClip, char
   *outDate, FPInt *ioDateLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, FPInt *ioDateLen
```

Output Parameters:

```
char *outDate, FPInt *ioDateLen
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function retrieves the creation date, in string format, of the C-Clip. The time is specified in UTC (Coordinated Universal Time, also known as GMT —Greenwich Mean Time). The creation data is based on the time of the primary cluster when the C-Clip was created with `FPClip_Create()`.

For example, February 21, 2004 is expressed as: `2004.02.21 10:46:32 GMT`

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `char *outDate`
  `outDate` is the buffer that will store the creation date of the C-Clip. This date will be truncated to the buffer length as specified by `ioDateLen`.
- `FPInt *ioDateLen`
  Input: The reserved length, in characters, of the `outDate` buffer.
  Output: The actual length of the date string, in characters, including the end-of-string character.

Example:

```
FPInt datesize;
datesize = MAX_DATE_SIZE;
char date[MAX_DATE_SIZE];
FPClip_GetCreationDate (myClip, date, &datesize);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_TAG_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_GetEBRClassName

Syntax:

```
FPClip_GetEBRClassName (const FPClipRef inClip,
   char* outClassName, FPInt* ioNameLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, char* outClassName,
FPInt* ioNameLen
```

Output Parameters:

```
FPInt* ioNameLen
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves the name of the event-based retention (EBR) class assigned to the C-Clip. This retention class was previously set by `FPClip_EnableEBRWithClass()` or `FPClip_TriggerEBREvent WithClass()`.

If an EBR retention class was specified using both the `FPClip_EnableEBRWithClass()` and `FPClip_TriggerEBREvent()` calls, the SDK returns the class with the longest retention period.

If no EBR retention class is set or if there is an explicit EBR retention period associated with the C-Clip that is longer than the period associated with the explicit EBR retention class, this function returns an empty string.

Note: For more information on event-based retention, refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `char* outClassName`
  The buffer to which the EBR retention class name is written.

- `FPInt* ioNameLen`
  Input: The reserved length, in characters, of the `outClassName` buffer.
  Output: The actual length of the retention class name, in characters, including the end-of-string character.

Example:

```
FPInt nameSize = MAX_NAME_SIZE;
char className[MAX_NAME_SIZE];
FPClip_GetEBRClassName (myClip, className, &NameSize);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_INVALID_NAME` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_GetEBREventTime

Syntax:

```
FPClip_GetEBREventTime (const FPClipRef inClip,
   char* outEBREventTime, FPInt* ioEBREventTimeLen)
```

Return Value: `void`

Input Parameters: `const FPClipRef inClip, FPInt* ioEBREventTimeLen`

Output Parameters: `char* outEBREventTime, FPInt* ioEBREventTimeLen`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function returns the event time set on a C-Clip when the event-based retention (EBR) event for that C-Clip was triggered. The time is specified in UTC (Coordinated Universal Time, also known as GMT —Greenwich Mean Time).

If the event time is not available, for example, if EBR is not enabled for the C-Clip or if it has not occurred yet, the SDK returns an empty string.

Note: For more information on event-based retention, refer to Chapter 5, *Data Retention,* in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `char* outEBREventTime`
  `outEBREventTime` is the memory buffer that will store the EBR event time. The time is specified in `YYYY.MM.DD hh:mm:ss` GMT format.
- `FPInt *ioEBREventTimeLen`
  Input: The reserved length, in characters, of the `outEBREventTime` buffer.
  Output: The actual length of the string, in characters, including the end-of-string character.

Example:

```
char vWaitingTime[256];
FPInt vEBREventTimeLen=256;
FPClip_GetEBREventTime (vClip, vWaitingTime,
   &vEBREventTimeLen);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_GetEBRPeriod

Syntax: `FPClip_GetEBRPeriod (const FPClipRef inClip)`

Return Value: `FPLong`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function returns the value (in seconds) of the event-based retention (EBR) period associated with a C-Clip. If this period was set (directly or indirectly) by both `FPClip_EnableEBRWithPeriod/Class()` and `FPClip_TriggerEBREventWithPeriod/Class()`, then it represents the longer of the two values. If neither an EBR period or class is set, for example, if EBR is not enabled for the C-Clip, the SDK returns `FP_NO_RETENTION_PERIOD` (`0`).

Note: For more information on event-based retention, refer to Chapter 5, *Data Retention* in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
vWaitingTime = FPClip_GetEBRPeriod (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_INVALID_NAME` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_GetName

Syntax:

```
FPClip_GetName (const FPClipRef inClip, char *outName,
   FPInt *ioNameLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, FPInt *ioNameLen
```

Output Parameters

```
char *outName, FPInt *ioNameLen
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves the name of the given C-Clip. The name is returned in `outName`.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `char *outName`
  `outName` is the buffer that will store the name of the C-Clip. The name is truncated if necessary to the buffer length as specified by `ioNameLen`.
- `FPInt *ioNameLen`
  Input: The reserved length, in characters, of the `outName` buffer.
  Output: The actual length of the name, in characters, including the end-of-string character. If this value is larger than the length of the provided buffer, then the full name was not returned.

Example:

```
FPInt namesize;
namesize = MAX_NAME_SIZE;
char name[MAX_NAME_SIZE];
FPClip_GetName (myClip, name, &namesize);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_TAG_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_GetNumBlobs

Syntax: `FPClip_GetNumBlobs (const FPClipRef inClip)`

Return Value: `FPInt`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function returns the number of blobs that are associated with the given C-Clip.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:
```
NumBlobs = FPClip_GetNumBlobs (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_TAG_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_GetNumTags

Syntax: `FPClip_GetNumTags (const FPClipRef inClip)`

Return Value: `FPInt`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function returns the number of application-specific tags that are defined in the given C-Clip. Only tags created with `FPTag_Create()` are taken into account. This function returns -1 when an error occurs.

Note: C-Clips created with an SDK version lower than 1.2 must be opened in tree mode in order to retrieve the number of tags, otherwise the error `FP_ATTR_NOT_FOUND_ERR` is returned.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
NumTags = FPClip_GetNumTags (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_GetPoolRef

Syntax: `FPClip_GetPoolRef (const FPClipRef inClip)`

Return Value: `FPPoolRef`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function is thread safe.

Description: This function returns the reference to the pool in which the given C-Clip has been opened.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
myPool = FPClip_GetPoolRef (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error

FPClip_GetRetentionClassName

Syntax:
```
FPClip_GetRetentionClassName (const FPClipRef inClipRef,
   char *outName, FPInt *ioNameLen)
```

Return Value:
```
void
```

Input Parameters:
```
const FPClipRef inClipRef, FPInt *ioNameLen
```

Output Parameters
```
char *outName, FPInt *ioNameLen
```

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function retrieves the name of the retention class for the given C-Clip as set by `FPClip_SetRetentionClass()`.

For more information on retention periods and classes, refer to the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClipRef`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `char *outName`
  `outName` is the buffer that will store the name of the retention class. The name will be truncated if necessary to the buffer length as specified by `ioNameLen`. `outName` is an empty string if the specified C-Clip has no retention class set.
- `FPInt *ioNameLen`
  Input: The reserved length, in characters, of the `outName` buffer.
  Output: The actual length of the name, in characters, including the end-of-string character.

Example:
```
FPInt nameSize = MAX_NAME_SIZE;
char className[MAX_NAME_SIZE];
FPClip_GetRetentionClassName (myClip, className,
   &NameSize);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)

FPClip_GetRetentionHold

Syntax: `FPClip_GetRetentionHold (const FPClipRef inClip)`

Return Value: `FPBool`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function determines the hold state of the C-Clip. If there is retention hold on the C-Clip, `FPClip_GetRetentionHold()` returns `True`, otherwise, it is `False`.

Note: For more information on retention hold and event-based retention (EBR), refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:
```
vOnRetentionHold = FPClip_GetRetentionHold (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_GetRetentionPeriod

Syntax: `FPClip_GetRetentionPeriod (const FPClipRef inClip)`

Return Value: `FPLong`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function returns the retention period, in seconds, of the given C-Clip. The retention period was set by `FPClip_SetRetentionPeriod()` or `FPClip_SetRetentionClass()`, or is the cluster's default retention period. Refer to *FPClip_SetRetentionPeriod* on page 1-72 and *FPClip_SetRetentionClass* on page 1-68 for more information.

For more information on retention periods and classes, refer to the *Centera Programmer's Guide*, P/N 069001127.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:
```
vNumSeconds = FPClip_GetRetentionPeriod(vClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_GetTotalSize

Syntax:

```
FPClip_GetTotalSize (const FPClipRef inClip)
```

Return Value:

`FPLong`

Input Parameters:

`const FPClipRef inClip`

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function returns the total size (in bytes) of all blobs associated with the given C-Clip.

Note: The returned size does not include the size of the CDF.

Parameters:

`const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
ClipSize = FPClip_GetTotalSize (myClip);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (internal error)
- `FP_TAG_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_IsEBREnabled

Syntax: `FPClip_IsEBREnabled (const FPClipRef inClip)`

Return Value: `FPBool`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function returns a Boolean value to indicate whether or not a C-Clip is enabled for event-based retention (EBR). If the C-Clip is EBR-enabled, the function call returns `True`, otherwise, it is `False`.

Note: For more information on how to enable EBR, refer to *FPClip_EnableEBRWithClass* on page 1-56 and *FPClip_EnableEBRWithPeriod* on page 1-58.

Note: For more information on event-based retention, refer to Chapter 5, *Data Retention*, in the *Centera Programmer's Guide*, P/N 069001127.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
FPClip_IsEBREnabled (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ADVANCED_RETENTION_DISABLED_ERR` (server error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_IsModified

Syntax: `FPClip_IsModified (const FPClipRef inClip)`

Return Value: `FPBool`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: This function returns the modification status of an open C-Clip. This function returns true if the C-Clip has been modified since it was created, opened, or written. This function returns false if the C-Clip is the same as the C-Clip that is written to the pool.

Note: Use this function to determine whether a C-Clip should be written to a pool. If the function returns false, then there is no need to write the C-Clip to the pool.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
if (FPClip_IsModified (myClip)) {
   FPClip_Write (myClip, myClipID);
}
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_ValidateRetentionClass

Syntax:

```
FPClip_ValidateRetentionClass (const FPClipRef inClip,
   const FPRetentionClassContextRef inContext)
```

Return Value: `FPBool`

Input Parameters: `const FPClipRef inClip, const FPRetentionClassContextRef inContext`

Concurrency Requirement: This function is thread safe.

Description: This function returns true if the retention class associated with the specified C-Clip (as set by `FPClip_SetRetentionClass()`) is defined in the specified retention-class context, and returns false otherwise.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.
- `const FPRetentionClassContextRef inContext`
  The reference to a retention class context as returned by `FPPool_GetRetentionClassContext()`.

Example:

```
FPBool classIsDefined;
classIsDefined = FPClip_ValidateRetentionClass(myClip,
   myRetentionClassContext);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

Clip Attribute Functions

This section describes the functions that define and operate on C-Clip attributes:

- `FPClip_GetDescriptionAttribute`
- `FPClip_GetDescriptionAttributeIndex`
- `FPClip_GetNumDescriptionAttributes`
- `FPClip_RemoveDescriptionAttribute`
- `FPClip_SetDescriptionAttribute`

The size of an attribute value is limited to 100 KB. The number of attributes allowed in a C-Clip is limited only by the maximum size of a C-Clip, which is 100 MB.

FPClip_GetDescriptionAttribute

Syntax:

```
FPClip_GetDescriptionAttribute (const FPClipRef inClip,
   const char *inAttrName, const char *outAttrValue,
   FPInt *ioAttrValueLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const char *inAttrName, FPInt
*ioAttrValueLen
```

Output Parameters:

```
const char *outAttrValue, FPInt *ioAttrValueLen
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves the value of the given attribute from the given C-Clip.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const char *inAttrName`
  `inAttrName` is the buffer that contains the name of the attribute for which the value is retrieved.
- `const char *outAttrValue`
  `outAttrValue` is the buffer that will store the attribute value.
- `FPInt *ioAttrValueLen`
  Input: The reserved length, in characters, of the `outAttrValue` buffer.
  Output: The actual length of the string, in characters, including the end-of-string character.

Example:

```
char vBuffer[1024];
FPInt l=sizeof(vBuffer);
FPClip_GetDescriptionAttribute(myClip, "company",
   vBuffer, &l);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)

FPClip_GetDescriptionAttributeIndex

Syntax:

```
FPClip_GetDescriptionAttributeIndex (const FPClipRef
   inClip, const FPInt inIndex, char *outAttrName, FPInt
   *ioAttrNameLen, char *outAttrValue, FPInt
   *ioAttrValueLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const FPInt inIndex, FPInt
*ioAttrNameLen, FPInt *ioAttrValueLen
```

Output Parameters:

```
char *outAttrName, FPInt *ioAttrNameLen, char
*outAttrValue, FPInt *ioAttrValueLen
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves the name and value of an attribute from the given C-Clip according to the given index.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const FPInt inIndex`
  The index number (zero based) of the attribute that has to be retrieved.
- `char *outAttrName`
  `outAttrName` is the buffer that will store the attribute name.
- `char *outAttrValue`
  `outAttrValue` is the buffer that will store the attribute value.
- `FPInt *ioAttrNameLen`
  Input: The reserved length, in characters, of the `outAttrName` buffer.
  Output: The actual length of the name, in characters, including the end-of-string character.

- `FPInt *ioAttrValueLen`
  Input: The reserved length, in characters, of the `outAttrValue` buffer.
  Output: The actual length of the value, in characters, including the end-of-string character.

Example:

```
char vName[256];
char vValue[256];
for (int i=0;
   i<FPClip_GetNumDescriptionAttributes(myClip); i++)
   {FPInt vNameLen = sizeof(vName);
    FPInt vValueLen = sizeof(vValue);
    FPClip_GetDescriptionAttributeIndex(myClip, i, vName,
   &vNameLen, vValue, &vValueLen);
   }
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)

FPClip_GetNumDescriptionAttributes

Syntax:

```
FPClip_GetNumDescriptionAttributes (const FPClipRef
   inClip)
```

Return Value:

`FPInt`

Input Parameters:

`const FPClipRef inClip`

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function returns the number of the user-defined and standard description attributes of the given C-Clip.

Parameters:

`const FPClipRef inClip`
The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
FPInt vNum;
vNum = FPClip_GetDescriptionAttributes(myClip);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPClip_RemoveDescriptionAttribute

Syntax:

```
FPClip_RemoveDescriptionAttribute (const FPClipRef
   inClip, const char *inAttrName)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const char *inAttrName
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function removes the given attribute from the CDF of the given C-Clip.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const char *inAttrName`
  `inAttrName` is the buffer that contains the name of the attribute that needs to be removed.

Example:

```
FPClip_RemoveDescriptionAttribute(myClip, "company");
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)

FPClip_SetDescriptionAttribute

Syntax:

```
FPClip_SetDescriptionAttribute (const FPClipRef inClip,
   const char *inAttrName, const char *inAttrValue)
```

Return Value:

```
void
```

Input Parameters:

```
const FPClipRef inClip, const char *inAttrName, const
char *inAttrValue
```

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function adds the given attribute (name-value pair) to the CDF of the given C-Clip.

You can also add user-defined attributes using the environment variable `CENTERA_CUSTOM_METADATA`. The SDK reads this variable during `FPClip_Create()`. Refer to *FPClip_AuditedDelete* on page 1-48 for more information.

Use `FPClip_GetDescriptionAttribute()` to read the user-defined attributes. Use `FPClip_GetDescriptionAttributeIndex()` to see which user-defined attributes and standard metadata attributes the CDF contains.

Parameters:

- `const FPClipRef inClip`
  The reference to a C-Clip that was opened by `FPClip_Open()` or `FPClip_Create()`.
- `const char *inAttrName`
  `inAttrName` is the buffer that contains the name of the attribute that needs to be added.
- `const char *inAttrValue`
  `inAttrValue` is the buffer that contains the value of the attribute that needs to be added. The maximum allowed attribute value size is 100 KB.

Note: To ensure compatibility with future SDK releases, attribute values should not contain control characters, such as newlines and tabs.

Example:

```
myClip = FPClip_Create(myPool, "test");
FPClip_SetDescriptionAttribute(myClip, "company",
   "com.acme");
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)

Clip Tag Functions

This section describes the functions that manipulate a single tag from a C-Clip. Because these functions operate on the level of a C-Clip—not on the level of a tag as the tag functions described in *Tag Functions* on page 1-123—they are listed as clip functions.

- `FPClip_FetchNext`
- `FPClip_GetTopTag`

FPClip_FetchNext

Syntax:

```
FPClip_FetchNext (const FPClipRef inClip)
```

Return Value:

`FPTagRef`

Input Parameters:

`const FPClipRef inClip`

Concurrency Requirement:

This function requires exclusive access to the C-Clip reference in memory.

Description:

This function returns the next tag in the tag structure. If this is the first call to the function, the first tag is returned. Call `FPClip_GetTopTag()` to restart at the first tag.

If the C-Clip was opened in tree mode, the traversal order is depth-first, which returns the tags in the same order as if the C-Clip was opened in flat mode.

This function returns `NULL` if the C-Clip has no tags, or if the last tag was already returned.

Be sure to close the tag with `FPTag_Close()` after you are done processing the tag to free allocated resources.

Parameters:

`const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
myClip = FPClip_Open(myPool, myClipID, FP_OPEN_FLAT);

While ((myTag = FPClip_FetchNext(myClip)) ! = 0)
{//...do something with the tag
  FPTag_Close(myTag);
}
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_TAGTREE_ERR` (internal error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPClip_GetTopTag

Syntax: `FPClip_GetTopTag (const FPClipRef inClip)`

Return Value: `FPTagRef`

Input Parameters: `const FPClipRef inClip`

Concurrency Requirement: This function requires exclusive access to the C-Clip reference in memory.

Description: The behavior of this function depends on whether the C-Clip is open in tree mode or flat mode. If you are creating or modifying tags, you must open the C-Clip in tree mode. Refer to *FPPool_Close* on page 1-12 for more information.

- Flat mode

  This function returns a reference to the first user tag (as created by `FPTag_Create()`) in the specified C-Clip. Call `FPClip_FetchNext()` to retrieve subsequent tags.

- Tree mode

  This function returns a reference to the top-level tag in a C-Clip.

  If the tag structure is empty, you can use the top tag as the parent tag to create the first user tag in the tree.

  If the tag structure is not empty, you can use the top tag as a starting point for further navigation. You can use `FPTag_GetFirstChild()`, then `FPTag_GetSibling()` and `FPTag_GetPrevSibling()` to retrieve tags hierarchically, or `FPClip_FetchNext()` to retrieve tags in sequential order.

Note: Unlike user tags, the top tag has no name, attributes, or associated blob. For example, you cannot call `FPTag_GetTagName()` on the top tag.

Be sure to close the tag with `FPTag_Close()` after you are done processing the tag to free allocated resources.

Parameters: `const FPClipRef inClip`
The reference to a C-Clip opened by `FPClip_Open()` or `FPClip_Create()`.

Example:

```
myTag = FPClip_GetTopTag (myClip);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` (zero) if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_TAGTREE_ERR` (internal error)
- `FP_POOLCLOSED_ERR` (program logic error)

Tag Functions

The tag functions operate at the level of a C-Clip tag. Tags are used to support self-describing content. The tag functions are subdivided into four categories based on their use:

- **Tag Handling Functions** — to manipulate tags
- **Tag Navigation Functions** — to navigate through the C-Clip tag structure
- **Tag Attribute Functions** — to manipulate tag attributes
- **Blob Handling Functions** — to manipulate blobs

Before you can perform a tag operation, you must first create or retrieve a tag. Close each tag after processing is complete.

Tag Handling Functions

This section describes the following functions that handle a tag within a C-Clip:

- `FPTag_Close`
- `FPTag_Copy`
- `FPTag_Create`
- `FPTag_Delete`
- `FPTag_GetBlobSize`
- `FPTag_GetClipRef`
- `FPTag_GetPoolRef`
- `FPTag_GetTagName`

FPTag_Close

Syntax: `FPTag_Close (const FPTagRef inTag)`

Return Value: `void`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function closes the given tag and frees all allocated resources. Note that calling this function on a tag that has already been closed may produce unwanted results.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPClip_GetTopTag()`, or `FPClip_FetchNext()`).

Example:
```
FPTag_Close (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_Copy

Syntax:

```
FPTag_Copy (const FPTagRef inTag, const FPTagRef
   inNewParent, const FPInt inOptions)
```

Return Value:

`FPTagRef`

Input Parameters:

`const FPTagRef inTag, const FPTagRef inNewParent, const FPInt inOptions`

Concurrency Requirement:

This function is thread safe.

Description:

This function creates a new tag and copies the given tag to the new destination tag. The destination tag does not have to be in the same C-Clip but must belong to a C-Clip that was open from the same pool object. When copying multiple children of the same tag, the order of the tags is preserved.

Note: Be sure to close the new tag after `FPTag_Copy()` has been called.

The result of this function is the same as performing multiple `FPTag_Create()` calls.

Note: This function is supported only if the C-Clip to which the tag belongs resides on the same cluster as the C-Clip to which the new destination tag belongs. If the C-Clips reside on different pools, the call returns `FP_OPERATION_NOT_SUPPORTED`.

Parameters:

- `const FPTagRef inTag`
  The reference to the tag that you want to copy.
- `const FPTag inNewParent`
  The reference to the new destination tag. To copy a tag, a reference to an existing destination tag is required.
- `const FPInt inOptions`
  You can use one or more of the following options:
  - `FP_OPTION_NO_COPY_OPTIONS` — Only the tag and its attributes are copied.
  - `FP_OPTION_COPY_BLOBDATA` — The tag attributes and the blob data are copied.

Note: The blob IDs are copied and no actual data is moved.

`FP_OPTION_COPY_CHILDREN` — The children of the tag are copied. You must specify this option if `inTag` is the top tag. The top tag itself is not copied. If `inTag` is a parent of `inNewParent`, then the error `FP_OUT_OF_BOUNDS_ERR` is returned.

Example: To make a full copy of one C-Clip to another:

```
vClip = FPClip_Open (vPoolRef, vNewID, FP_OPEN_ASTREE);
vTop = FPClip_GetTopTag (vClip);
vClip1 = FPClip_Create (vPoolRef, "copy of C-Clip");
vTop1 = FPClip_GetTopTag (vClip1);
FPTag_Copy (vTop, vTop1, FP_OPTION_COPY_BLOBDATA |
   FP_OPTION_COPY_CHILDREN);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)

FPTag_Create

Syntax:

```
FPTag_Create (const FPTagRef inParent, const char
   *inName)
```

Return Value:

```
FPTagRef
```

Input Parameters:

```
const FPTagRef inParent, const char *inName
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function creates a new tag within a C-Clip that has been opened in tree mode (refer to `FPClip_Open()`), and returns a reference to the new tag. The number of tags in a C-Clip is restricted only by the maximum size of a C-Clip: 100 MB.

Note: A reference to a parent tag is required to create a new tag.

Parameters:

- `const FPTagRef inParent`
  The reference to the parent tag of the new tag that you are creating.
- `const char *inName`
  `inName` is the buffer that stores the name of the new tag. The value cannot be `NULL`. The characters accepted by this function are ASCII characters in the Set [a-zA-Z0-9_-.]. The first character must be a letter or an underscore "_". If your application requires other characters, use `FPTag_CreateW()`.

Note: The name must be XML compliant and cannot start with the prefix "xml" or "eclip".

Example:

```
myTag = FPTag_Create (Parent, "tagname");
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_INVALID_NAME` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_Delete

Syntax: `FPTag_Delete (const FPTagRef inTag)`

Return Value: `void`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function deletes a tag (and all children of the tag) in the tag structure of a C-Clip.

If the C-Clip has been opened in flat mode, this function returns `FP_OPERATION_NOT_SUPPORTED`. If the function tries to delete a top tag, the function returns `FP_TAG_READONLY_ERR`.

Note: After a successful deletion, the system deallocates the memory for the tag and `inTag` becomes invalid. Any function call to the tag (for example the `FPTag_Close()` function) results in an `FP_WRONG_REFERENCE_ERR` error.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, or `FPTag_GetPrevSibling()`).

Example:

```
FPTag_Delete (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetBlobSize

Syntax: `FPTag_GetBlobSize (const FPTagRef inTag)`

Return Value: `FPLong`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the total size, in bytes, of the blob content associated with the tag. If the tag has no associated blob content, the return value is -1.

Note: `FPTag_GetBlobSize()` best supports C-Clips that are opened in tree mode.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPClip_GetTopTag()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).

Example:

```
BlobSize = FPTag_GetBlobSize (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPTag_GetClipRef

Syntax: `FPTag_GetClipRef (const FPTagRef inTag)`

Return Value: `FPClipRef`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the reference to the C-Clip in which the given tag was opened.

Parameters: `const FPTagRef inTag`
The reference to a tag that any of the tag functions have navigated to or created.

Example:
```
myClip = FPTag_GetClipRef (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetPoolRef

Syntax: `FPTag_GetPoolRef (const FPTagRef inTag)`

Return Value: `FPPoolRef`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the reference to the pool in which the given tag was opened.

Parameters: `const FPTagRef inTag`
The reference to a tag that any of the tag functions have opened or created.

Example:
```
myPool = FPTag_GetPoolRef (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetTagName

Syntax:

```
FPTag_GetTagName (const FPTagRef inTag, char *outName,
   FPInt *ioNameLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, FPInt *ioNameLen
```

Output Parameters:

```
char *outName, FPInt *ioNameLen
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves the name of the given tag.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPClip_FetchNext()`, `FPTag_Copy()`, or `FPCLIP_GetTopTag()` used in flat mode).
- `char *outName`
  `outName` is the buffer that will store the name of the tag. The name will be truncated to the buffer length as specified by `ioNameLen`.
- `FPInt *ioNameLen`
  Input: The reserved length, in characters, of the `outName` buffer.
  Output: The actual length of the name string, in characters, including the end-of-string character.

Example:

```
FPInt namesize;
namesize = MAX_NAME_SIZE;
char name[MAX_NAME_SIZE];
FPTag_GetTagName (myTag, name, &namesize);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)

Tag Navigation Functions

The two ways to navigate through the tag structure of a C-Clip are:

- **Hierarchically** — Open the C-Clip as a tree structure (refer to `FPClip_Open()`). The C-Clip is opened in read/write mode.
- **Sequentially** — Open the C-Clip as a flat structure (refer to `FPClip_Open()`). The C-Clip is opened in read-only mode. This option avoids the use of large memory buffers and is useful for reading C-Clips that do not fit into memory.

This section describes the following functions that handle tag navigation within a C-Clip:

- `FPTag_GetFirstChild`
- `FPTag_GetParent`
- `FPTag_GetPrevSibling`
- `FPTag_GetSibling`

Figure 1-1 shows how the tags are structured hierarchically and how the tag navigation functions operate if the C-Clip opens in tree mode.

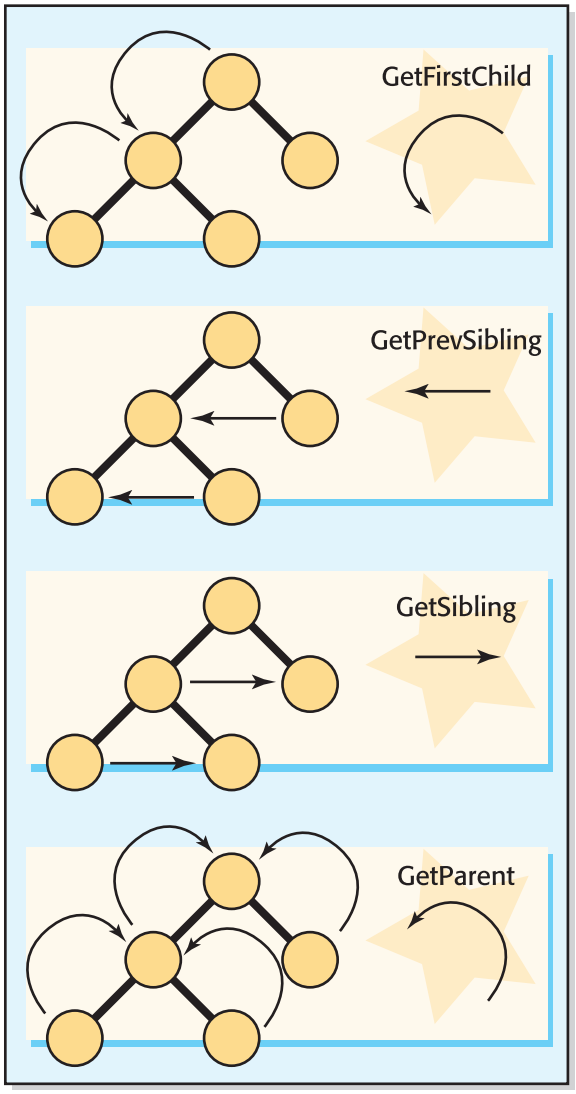

Figure 1-1 Tag Structure and Navigation

FPTag_GetFirstChild

Syntax: `FPTag_GetFirstChild (const FPTagRef inTag)`

Return Value: `FPTagRef`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the first child tag of the given tag. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`).

A child tag is the tag that is one level down from the given tag in the tag hierarchy (refer to Figure 1-1). This function returns `NULL` if no child tag can be found.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPClip_GetTopTag()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, or `FPTag_GetPrevSibling()`).

Example:

```
myChildTag = FPTag_GetFirstChild (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetParent

Syntax: `FPTag_GetParent (const FPTagRef inTag)`

Return Value: `FPTagRef`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the parent tag of the given tag. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`).

A parent tag is one level up from the given tag in the tag hierarchy (refer to Figure 1-1 on page 1-136). This function returns `NULL` if the system cannot find a parent tag.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, or `FPTag_GetPrevSibling()`).

Example:

```
myParent = FPTag_GetParent (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetPrevSibling

Syntax:

```
FPTag_GetPrevSibling (const FPTagRef inTag)
```

Return Value:

`FPTagRef`

Input Parameters:

`const FPTagRef inTag`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns the previous sibling tag of the given tag. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`).

A previous sibling tag is at the same level as the given tag in the tag hierarchy of the C-Clip but opposite to the sibling tag that is retrieved by `FPTag_GetSibling` (refer to Figure 1-1 on page 1-136).

This function returns `NULL`, if the system cannot find a sibling tag.

Parameters:

`const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPTag_GetFirstChild()`.

Example:

```
mySibling = FPTag_GetPrevSibling (myTag);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetSibling

Syntax: `FPTag_GetSibling (const FPTagRef inTag)`

Return Value: `FPTagRef`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the sibling tag of the given tag. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`).

A sibling tag is at the same level as the given tag in the tag hierarchy of the C-Clip but opposite to the previous sibling tag that is retrieved by `FPTag_GetPrevSibling()` (refer to Figure 1-1 on page 1-136).

This function returns `NULL` if the system cannot find a sibling tag.

To go back to the tag where you started, use `FPTag_GetPrevSibling()` or `FPClip_GetParent()` and `FPClip_GetFirstChild()`.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPTag_GetFirstChild()`.

Example:

```
mySibling = FPTag_GetSibling (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

Tag Attribute Functions

There are two sets of tag attribute functions:

- Functions to set an attribute value (set attribute functions)
- Functions to get an attribute value (get attribute functions)

With the set attribute functions you can assign a specified value to a tag attribute. If the attribute does not exist yet, the function creates the attribute. If the attribute exists, the function overwrites the current value of that attribute. The get attribute functions are used to retrieve the assigned attribute values.

The size of an attribute value is limited to 100 KB. The number of attributes allowed in a tag is limited only by the maximum size of a C-Clip, which is 100 MB.

This section describes the following functions that enable the setting and retrieval of tag attribute values:

- `FPTag_GetBoolAttribute`
- `FPTag_GetIndexAttribute`
- `FPTag_GetLongAttribute`
- `FPTag_GetNumAttributes`
- `FPTag_GetStringAttribute`
- `FPTag_RemoveAttribute`
- `FPTag_SetBoolAttribute`
- `FPTag_SetLongAttribute`
- `FPTag_SetStringAttribute`

FPTag_GetBoolAttribute

Syntax:

```
FPTag_GetBoolAttribute (const FPTagRef inTag, const char
   *inAttrName)
```

Return Value: `FPBool`

Input Parameters: `const FPTagRef inTag, const char *inAttrName`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function returns a Boolean attribute of an existing tag of a C-Clip.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is the buffer containing the name of the attribute.

Example:

```
FPTag_GetBoolAttribute (myTag, "attribute_name");
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)

FPTag_GetIndexAttribute

Syntax:

```
FPTag_GetIndexAttribute (const FPTagRef inTag, const
   FPInt inIndex, char *outAttrName, FPInt
   *ioAttrNameLen, char *outAttrValue, FPInt
   *ioAttrValueLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const FPInt inIndex, FPInt
*ioAttrNameLen, FPInt *ioAttrValueLen
```

Output Parameters:

```
char *outAttrName, FPInt *ioAttrNameLen, char
*outAttrValue, FPInt *ioAttrValueLen
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function returns an attribute name and value of an existing tag in a C-Clip using the given index number.

Parameters:

- `FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `FPInt inIndex`
  `inIndex` is the index number (zero based) of the tag attribute that has to be retrieved.
- `char *outAttrName`
  `outAttrName` is the buffer that will hold the name of the attribute. The name will be truncated to the buffer length as specified by `ioAttrNameLen`.
- `FPInt *ioAttrNameLen`
  Input: The reserved length, in characters, of the `outAttrName` buffer.
  Output: The actual length of the attribute name, in characters, including the end-of-string character.

- `char *outAttrValue`
  `outAttrValue` is the buffer that will hold the value of the attribute. The value will be truncated to the buffer length as specified by `ioAttrValueLen`.
- `FPInt *ioAttrValueLen`
  Input: The reserved length, in characters, of the `outAttrValue` buffer.
  Output: The actual length of the attribute value, in characters, including the end-of-string character.

Example:

```
char TagAttrName[MAX_NAME_SIZE];
char TagAttrValue[MAX_NAME_SIZE];
NumAttributes = FPTag_GetNumAttributes(Tag);
if (FPPool_GetLastError() != 0)
   handle error...
for (i = 0; i < NumAttributes; i++)
{
   AttrNameSize = MAX_NAME_SIZE;
   AttrValueSize = MAX_NAME_SIZE;
   FPTag_GetIndexAttribute(Tag, i,
        TagAttrName, &AttrNameSize,
        TagAttrValue, &AttrValueSize);
if (FPPool_GetLastError() != 0)
   handle error
printf("Attribute #%d has name \"%s\" and value
   \"%s\".\n",
   i, TagAttrName, TagAttrValue);
}
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetLongAttribute

Syntax:

```
FPTag_GetLongAttribute (const FPTagRef inTag, const char
   *inAttrName)
```

Return Value:

`FPLong`

Input Parameters:

`const FPTagRef inTag, const char *inAttrName`

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function returns an `FPLong` attribute of an existing tag of a C-Clip.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from the functions `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is the buffer containing the name of the attribute.

Example:

```
myValue = FPTag_GetLongAttribute (myTag,
   "attribute_name");
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)

FPTag_GetNumAttributes

Syntax: `FPTag_GetNumAttributes (const FPTagRef inTag)`

Return Value: `FPInt`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function returns the number of attributes in a tag.

Parameters: `const FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPClip_GetTopTag()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).

Example:
```
NumAttrs = FPTag_GetNumAttributes (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_GetStringAttribute

Syntax:

```
FPTag_GetStringAttribute (const FPTagRef inTag, const
   char *inAttrName, char *outAttrValue, FPInt
   *ioAttrValueLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const char *inAttrName, FPInt
*ioAttrValueLen
```

Output Parameters:

```
char *outAttrValue, FPInt *ioAttrValueLen
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves a string attribute of an existing tag in a C-Clip and returns the value to a buffer with a specified length.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is the buffer containing the attribute name.
- `char *outAttrValue`
  `outAttrValue` is the buffer that will hold the attribute value. The value will be truncated to the buffer length as specified by `ioAttrValueLen`.
- `FPInt *ioAttrValueLen`
  Input: The reserved length, in characters, of the `outAttrValue` buffer.
  Output: The actual length of the attribute value, in characters, including the end-of-string character.

Example:

```
char outAttrValue[MAX_NAME_SIZE];
namesize = MAX_NAME_SIZE;
FPTag_GetStringAttribute (myTag, "name", outAttrValue,
   &namesize);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_ATTR_NOT_FOUND_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)

FPTag_RemoveAttribute

Syntax:

```
FPTag_RemoveAttribute (const FPTagRef inTag, const char
   *inAttrName)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const char *inAttrName
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function removes an attribute from a tag. The C-Clip containing the tag must have been opened in tree mode (refer to `FPClip_Open()`).

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is the buffer containing the name of the attribute that has to be removed.

Example:

```
FPTag_RemoveAttribute (myTag, "attribute_name");
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_SetBoolAttribute

Syntax:

```
FPTag_SetBoolAttribute (const FPTagRef inTag, const char
   *inAttrName, const FPBool inAttrValue)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const char *inAttrName, const
FPBool inAttrValue
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function sets a Boolean attribute for an existing tag in an opened C-Clip. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`). This function requires a reference to the tag that you want to update (`inTag`), the attribute name (`inAttrName`) and the attribute value (`inAttrValue`).

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is the buffer that will hold the name of the attribute to be created or updated.

  Note: The name must be XML compliant.

- `const FPBool inAttrValue`
  `inAttrValue` is the value of the attribute to be assigned.

Example:

```
FPBool inAttrValue;
inAttrValue = TRUE;
FPTag_SetBoolAttribute (myTag, "name", inAttrValue);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_INVALID_NAME` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_SetLongAttribute

Syntax:

```
FPTag_SetLongAttribute (const FPTagRef inTag, const char
   *inAttrName, const FPLong inAttrValue)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const char *inAttrName, const
FPLong inAttrValue
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function sets an `FPLong` attribute of the given tag in an open C-Clip. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`). This function requires a reference to the tag that has to be updated (`inTag`), the attribute name (`inAttrName`), and the attribute value (`inAttrValue`).

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is a string containing the name of the attribute to be created or updated.

  Note: The name must be XML compliant.

- `const FPLong inAttrValue`
  `inAttrValue` contains the value of the attribute to be assigned.

Example:

```
FPLong inAttrValue;
inAttrValue = 100;
FPTag_SetLongAttribute (myTag, "name", inAttrValue);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_INVALID_NAME` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

FPTag_SetStringAttribute

Syntax:

```
FPTag_SetStringAttribute (const FPTagRef inTag, const
   char *inAttrName, const char *inAttrValue)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const char *inAttrName, const char
*inAttrValue
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function sets a string attribute of the given tag in an opened C-Clip. The C-Clip must have been opened in tree mode (refer to `FPClip_Open()`). This function requires a reference to the tag that has to be updated (`inTag`), the attribute name (`inAttrName`), and the attribute value (`inAttrValue`).

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const char *inAttrName`
  `inAttrName` is the buffer containing the name of the attribute to be created or updated.

  Note: The name must be XML compliant.

- `const char *inAttrValue`
  `inAttrValue` contains the value of the attribute that will be assigned. The value cannot be `NULL` or an empty string. If the value is larger than 100 KB, EMC recommends writing the value as a separate blob to the pool in order to increase performance. The maximum allowed attribute value size is 100 KB.

Note: To ensure compatibility with future SDK releases, attribute values should not contain control characters, such as newlines and tabs.

Example:

```
FPTag_SetStringAttribute (myTag, "name", "value");
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_INVALID_NAME` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_SECTION_NOT_FOUND_ERR` (internal error)
- `FP_OUT_OF_BOUNDS_ERR` (program logic error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)

Blob Handling Functions

This section describes the following functions that manipulate a blob (a tag referring to actual data):

- `FPTag_BlobExists`
- `FPTag_BlobRead`
- `FPTag_BlobReadPartial`
- `FPTag_BlobWrite`
- `FPTag_BlobWritePartial`

FPTag_BlobExists

Syntax: `FPTag_BlobExists (const FPTagRef inTag)`

Return Value: `FPInt`

Input Parameters: `const FPTagRef inTag`

Concurrency Requirement: This function is thread safe.

Description: This function checks if the blob data of the given tag exists. If the blob data of the given tag exists, the function returns 1. If the blob data is segmented then all segments must exist. If the blob data does not exist, the function returns 0. If the tag has no associated blob data, the function returns -1.

Exists operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Note: The server allows the application to perform this call if the server capability "exist" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Parameters: `FPTagRef inTag`
The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPClip_FetchNext()`, or `FPTag_GetParent()`).

Example:

```
FPTag_BlobExists (myTag);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_TAGCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPTag_BlobRead

Syntax:

```
FPTag_BlobRead (const FPTagRef inTag, const FPStreamRef
   inStream, const FPLong inOptions)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const FPStreamRef inStream, const
FPLong inOptions
```

Concurrency Requirement:

Concurrent threads cannot operate on the same stream.

Description:

This function retrieves the blob data from the pool and writes it to the stream object (refer to *Stream Functions* on page 1-189).

Note: Refer to *Stream Creation Functions* on page 1-193 for more information on how a stream is opened.

`FPTag_BlobRead()` leaves the marker at the end of the stream. The stream does not have to support marking. If the operation fails, the operation continues from the point where it failed.

Note: Due to server limitations, the stream should not exceed 1 minute per iteration to store the data received from the server. This means the `completeProc` callback should not take longer than 1 minute to execute.

This function gets the Content Address from the tag, opens the blob, reads the data in chunks of 16 Kbyte, writes the bytes to the stream, and closes the blob.

Note: The server allows the application to perform this call if the server capability "read" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Read operations fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPClip_FetchNext()`, or `FPTag_GetParent()`).
- `const FPStreamRef inStream`
  The reference to a stream (as returned from the functions `FPStream_CreateXXX()` or `FPStream_CreateGenericStream()`).
- `const FPLong inOptions`
  Reserved for future use. Specify `FP_OPTION_DEFAULT_OPTIONS`.

Example:

```
FPTag_BlobRead (myTag, myStream,
   FP_OPTION_DEFAULT_OPTIONS);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPTag_BlobReadPartial

Syntax:

```
FPTag_BlobReadPartial (const FPTagRef inTag, const
   FPStreamRef inStream, const FPLong inOffset, const
   FPLong inReadLength, const FPLong inOptions)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const FPStreamRef inStream, const
FPLong inOffset, const FPLong inReadLength, const FPLong
inOptions
```

Concurrency Requirement:

Concurrent threads cannot operate on the same stream.

Description:

This function retrieves the blob data from the pool and writes the data to a stream object that the application provides (refer to *Stream Functions* on page 1-189). This function reads the data in chunks of 16 Kbyte.

Note: The server allows the application to perform this call if the server capability "read" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

This function gets the Content Address from the tag, opens the blob, starts reading the blob packet as specified by the given offset, writes the specified bytes to the stream, and closes the blob.

Note: If the offset tries to read past the end of the blob, then no data is added to the output stream and the function returns `ENOERR`. Use `FPTag_GetBlobSize()` to verify that you are not reading past the end of the blob.

Refer to *Stream Creation Functions* on page 1-193 for more information on how a stream is opened.

Note: Due to server limitations, the stream should not exceed 1 minute per iteration to store the data received from the server. This means the `completeProc` callback should not take longer than 1 minute to execute.

Read operations fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPClip_FetchNext()`, or `FPTag_GetParent()`).
- `const FPStreamRef inStream`
  The reference to a stream (as returned from the functions `FPStream_CreateXXX()` or `FPStream_CreateGenericStream()`).
- `const FPLong inOffset`
  The starting offset of the read operation.
- `const FPLong inReadLength`
  The length in bytes of the data chunk to be read. Specify -1 if you want to read all data from the offset to the end of the blob.
- `const FPLong inOptions`
  Reserved for future use. Specify `FP_OPTION_DEFAULT_OPTIONS`.

Example:

Read 8 Kbytes of the blob, starting at the first byte.

```
FPTag_BlobReadPartial(myTag, myStream, 0, 8192,
FP_OPTION_DEFAULT_OPTIONS);
```

Read 8 Kbytes of the blob, starting at offset 32 Kbytes.

```
FPTag_BlobReadPartial(myTag, myStream, 32768, 8192,
FP_OPTION_DEFAULT_OPTIONS);
```

Read the entire blob, equivalent to `FPTag_BlobRead(myTag, myStream, FP_OPTION_DEFAULT_OPTIONS)`.

```
FPTag_BlobReadPartial(myTag, myStream, 0, -1,
FP_OPTION_DEFAULT_OPTIONS);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)

FPTag_BlobWrite

Syntax:

```
FPTag_BlobWrite (const FPTagRef inTag, const FPStreamRef
   inStream, const FPLong inOptions)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const FPStreamRef inStream, const
FPLong inOptions
```

Concurrency Requirement:

Concurrent threads cannot operate on the same stream.

Description:

This function writes blob data to the pool from a stream object that the application provides (refer to the section *Stream Functions* on page 1-189).

This function supports two methods for storing data:

- Linked data — If the data size is larger than the embedding threshold (refer to *FPPool_SetGlobalOption* on page 1-38) or you specified the `FP_OPTION_LINK_DATA` option, the function creates a new attribute for the specified tag, opens a new blob, reads bytes from the stream object, writes the data to the blob, closes the blob, and sets the blob ID (Content Address) as the attribute value. By default, the embedding threshold is zero (0), so no data embedding occurs.
- Embedded data — If the data size is smaller than the embedding threshold (refer to *FPPool_SetGlobalOption* on page 1-38) or you specified the `FP_OPTION_EMBED_DATA` option, the function creates an attribute for the specified tag, reads bytes from the stream object, encodes the data as a base64 string, and sets this string as the value of the attribute. The function also calculates and stores the blob ID (Content Address) as the value of another tag attribute. For more information on embedding data, refer to the *Centera Programmer's Guide*, P/N 069001127).

The storage method is transparent to the client application—the blob functions (for example, `FPTag_BlobRead()`) behave identically for embedded and linked data.

Parameters that you specify determine whether the Content Address (CA) is calculated before or while sending the data to the pool. By default, the CA is calculated by the client while the data is being sent.

Ensure that the C-Clip to which the tag belongs has been opened in tree mode. A C-Clip opened in flat mode is read-only.

Note: The server allows the application to perform this call if the server capability "write" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Write operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

If collision avoidance is enabled at pool level, refer to *FPPool_SetClipID* on page 1-35, this call uses an additional blob discriminator during write and read operations of the blob. If you want to disable collision avoidance for this call—to be compatible with a Centera v1.2 cluster—use `FP_OPTION_DISABLE_COLLISION_AVOIDANCE`. If collision avoidance is disabled at pool level, you can enable it for this call using `FP_OPTION_ENABLE_COLLISION_AVOIDANCE`.

If this function is used to restore data from a stream to the pool, the data is only exported to the pool if `inTag` already has data associated with it and the stream data is the same.

Note: Due to server limitations, the stream should not exceed 1 minute per iteration to retrieve the data for transfer. In practice this means the `prepareBufferProc` callback should not take longer than 1 minute to execute.

If the generic stream returns more data than requested, the SDK enlarges the buffer to accommodate the additional data. If the stream cannot provide the data within 1 minute, set `mTransferLen` to `-1` and `mAtEOF` to false, which forces `FPTag_BlobWrite()` to issue a keep-alive packet to the server.

Segments are exported to the server as if they are different blobs. If an error occurs during the write operation, the function retries. The retry operation only works properly if the stream supports marking (and

goes back to a previous position which is the beginning of the current segment). If the stream does not support marking to a previous position, the write operation is restarted from the beginning of the stream.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPClip_FetchNext()`, or `FPTag_GetParent()`).
- `const FPStreamRef inStream`
  The reference to an input stream (as returned from the `FPStream_Create`*`XXX`*`()` functions or `FPStream_CreateGenericStream()`). Refer to *Stream Functions* on page 1-189.
- `const FPLong inOptions`
  A variety of options that control the writing of the blob. You can specify options from different option sets by using the OR operator ('OR'ing the values together).
  - ID Calculation — Identifies how the content address should be calculated. You must use one of the following options:
    - `FP_OPTION_CLIENT_CALCID` — The client calculates the content address before sending the data to the cluster. The client does not send the data if the cluster already contains identical data. Consider using this option when writing small files (smaller than 10 MB) and identical data is likely to exist on the cluster. The provided stream must support marking. All non-generic streams support marking; a generic stream may or may not support marking.
    - `FP_OPTION_CLIENT_CALCID_STREAMING` — The client calculates the content address while sending the data to the server. The client sends the data even if the cluster already contains the data. This option is equivalent to `FP_OPTION_DEFAULT_OPTIONS`. Consider using this option when writing large files (10 MB or larger), when using many threads, or when identical data is unlikely to exist on the cluster.

      You can convert `CLIENT_CALCID_STREAMING` to operate in the `SERVER_CALCID_STREAMING` mode by setting the option `FP_OPTION_DISABLE_CLIENT_STREAMING` in `FPPool_SetGlobalOption` or by setting it to `True` as an

environment variable. If the latter, the change does not require a recompilation of application code. The SDK then handles and processes all references to `CLIENT_CALCID_STREAMING` as `SERVER_CALCID_STREAMING`.

  - `FP_OPTION_SERVER_CALCID_STREAMING` — The Centera server calculates the content address as the application server sends the data. There is no client check of the blob data before it is streamed to the cluster.

- Collision Avoidance — Specifies whether collision avoidance is enabled for this call to `FPTag_BlobWrite()`. Call `FPPool_SetIntOption()` to control collision avoidance at the pool level. Collision avoidance causes a unique blob ID to be generated for this content. Note that enabling collision avoidance disables Centera's single-instance storage feature. For more information refer to the *Centera Programmer's Guide*, P/N 069001127, and *FPPool_SetClipID* on page 1-35. Option choices are:
  - `FP_OPTION_ENABLE_COLLISION_AVOIDANCE` — Enables collision avoidance.
  - `FP_OPTION_DISABLE_COLLISION_AVOIDANCE` — Disables collision avoidance.
- Embedded Data — Specifies whether the data is stored in the CDF (embedded data), or as a separate blob with only the CA stored in the CDF (linked data). Choices are:
  - `FP_OPTION_EMBED_DATA` — Embed the data in the CDF. The maximum data size that can be embedded is 100 KB.

Note: If your program attempts to write an embedded blob that is too large, and the `FP_OPTION_EMBED_DATA` has been set, then an `FP_PARAM_ERR` is returned from the call, and the blob is not written.
To avoid losing data, ensure that your program checks for this error and that you are not attempting to write data in excess of 100 KB.
If the call fails, rewrite the blob as a linked blob.

  - `FP_OPTION_LINK_DATA` — Do not embed the data in the CDF.

  These options override the default embedded-data threshold (refer to *FPPool_SetGlobalOption* on page 1-38).

Example: The following example calculates the Content Address on the client while sending the data to the cluster and enables collision avoidance.

```
FPTag_BlobWrite (myTag, myStream,
   FP_OPTION_CLIENT_CALCID_STREAMING |
   FP_OPTION_ENABLE_COLLISION_AVOIDANCE);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_DUPLICATE_FILE_ERR` (internal error)
- `FP_NO_POOL_ERR` (network error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_MULTI_BLOB_ERR` (program logic error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_ACK_NOT_RCV_ERR` (server error)
- `FP_BLOBIDFIELD_ERR` (server error)
- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_SERVER_NO_CAPACITY_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPTag_BlobWritePartial

Syntax:

```
FPTag_BlobWritePartial (const FPTagRef inTag, const
   FPStreamRef inStream, const FPLong inOptions, const
   FPLong inSequenceID)
```

Return Value:

```
void
```

Input Parameters:

```
const FPTagRef inTag, const FPStreamRef inStream, const
   FPLong inOptions, const FPLong inSequenceID
```

Concurrency Requirement:

Concurrent threads cannot operate on the same stream.

Description:

This function can write blob data to the pool from a stream that the application provides (refer to the section *Stream Functions* on page 1-189). Applications can use multiple threads for the write operation and append data to existing data contained within an existing tag. This function uses the slicing-by-size model to write randomly accessed data. For more information on blob-slicing, refer to the *Centera Programmer's Guide*, P/N 069001127.

`FPTag_BlobWritePartial()` does not operate in conjunction with any of the embedded options. If you include `FP_OPTION_EMBED_DATA` in the inOptions parameter, an error occurs. Similarly, if you set `FP_OPTION_EMBEDDED_DATA_THRESHOLD`, the function ignores it.

Both `FPTag_BlobRead()` and `FPTag_BlobReadPartial()` can read this data transparently, which precludes the need for additional application coding.

The parameters you specify determine whether the Content Address (CA) is calculated before or while sending the data to the pool. By default, the CA is calculated by the client while the data is being sent. Ensure that the C-Clip to which the tag belongs has been opened in tree mode. A C-Clip opened in flat mode is read-only.

Note: The server allows the application to perform this call if the server capability "write" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Refer to *FPPool_GetCapability* on page 1-13 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *Centera Online Help*, P/N 300-002-656.

Write operations do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

If collision avoidance is enabled at pool level, refer to *FPPool_SetClipID* on page 1-35, this call uses an additional blob discriminator during write and read operations of the blob. If you want to disable collision avoidance for this call—to be compatible with a Centera v1.2 cluster—use `FP_OPTION_DISABLE_COLLISION_AVOIDANCE`. If collision avoidance is disabled at pool level, you can enable it for this call using `FP_OPTION_ENABLE_COLLISION_AVOIDANCE`.

Note: You cannot use `FPTag_BlobWritePartial()` to restore data to a Centera.

Note: Due to server limitations, the stream should not exceed 1 minute per iteration to retrieve the data for transfer. In practice this means the `prepareBufferProc` callback should not take longer than 1 minute to execute.

If the generic stream returns more data than requested, the SDK enlarges the buffer to accommodate the additional data. If the stream cannot provide the data within 1 minute, set `mTransferLen` to `-1` and `mAtEOF` to false, which forces `FPTag_BlobWritePartial()` to issue a keep-alive packet to the server.

Segments are exported to the server as if they are different blobs. If an error occurs during the write operation, the function retries. The retry operation only works properly if the stream supports marking (and goes back to a previous position which is the beginning of the current segment). If the stream does not support marking to a previous position, the write operation is restarted from the beginning of the stream.

Note: If you are not using `FPTag_BlobWritePartial` to perform multithreading, use `FPTag_BlobWrite` instead.

Parameters:

- `const FPTagRef inTag`
  The reference to a tag (as returned from `FPTag_Create()`, `FPTag_GetParent()`, `FPTag_GetFirstChild()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, or `FPClip_FetchNext()`).
- `const FPStreamRef inStream`
  The reference to an input stream (as returned from the `FPStream_Create`*`XXX`*`()` functions or `FPStream_CreateGenericStream()`). Refer to *Stream Functions* on page 1-189.
- `const FPLong inOptions`
  A variety of options that controls the writing of the blob. You can specify options from different option sets by using the OR operator ('OR'ing the values together).
  - ID Calculation — Identifies how the Content Address should be calculated. You must use one of the following options:
    - `FP_OPTION_CLIENT_CALCID` — The client calculates the address before sending the data to the cluster. The client does not send the data if the cluster already contains identical data. Consider using this option when writing small files (smaller than 10 MB) and identical data is likely to exist on the cluster. The provided stream must support marking. All non-generic streams support marking; a generic stream may or many not support marking.
    - `FP_OPTION_CLIENT_CALCID_STREAMING` — The client calculates the address while sending the data to the server. The client sends the data even if the cluster already contains the data. This option is equivalent to `FP_OPTION_DEFAULT_OPTIONS`. Consider using this option when writing large files (10 MB or larger), when using many threads, or when identical data is unlikely to exist on the cluster.

      You can convert `CLIENT_CALCID_STREAMING` to operate in the `FP_OPTION_SERVER_CALCID_STREAMING` mode by setting the option `FP_OPTION_DISABLE_CLIENT_STREAMING` in `FPPool_SetGlobalOption` or by setting it to `True` as an environment variable. If the latter, the change does not require a recompilation of application code. The SDK then handles and processes all references to `CLIENT_CALCID_STREAMING` as `SERVER_CALCID_STREAMING`.

  - FP_OPTION_SERVER_CALCID_STREAMING — The Centera server calculates the content address as the application server sends the data.

- Collision Avoidance — Specifies whether collision avoidance is enabled for this call to FPTag_BlobWritePartial(). Call FPPool_SetIntOption() to control collision avoidance at the pool level. Collision avoidance causes a unique blob ID to be generated for this content. Note that enabling collision avoidance disables Centera's single-instance storage feature. For more information refer to the *Centera Programmer's Guide*, P/N 069001127, and *FPPool_SetClipID* on page 1-35. Option choices are:
  - FP_OPTION_ENABLE_COLLISION_AVOIDANCE — Enables collision avoidance.
  - FP_OPTION_DISABLE_COLLISION_AVOIDANCE — Disables collision avoidance.

◆ const FPLong inSequenceID
The identifier that determines the sequence of the written data when the same inTag is used by one or more threads. This sequence sets the order in which the data is to be read back from Centera for the purpose of reassembling the blob segments. This read-back starts from the lowest ID to the highest ID.

If data is being written to a tag with existing data, it automatically is written to the end of the existing data.

An inSequenceID must be greater than or equal to zero. Threads that are using the same inTag cannot have duplicate sequence IDs. Each ID must be unique, otherwise, an FPParameterException is thrown.

Although sequence IDs do not need to be adjacent to one another, a single sequence ID cannot be reused within a tag during the same FPClip_Open/FPClip_Close operation. It can, however, be reused in that tag if reopened in another API call.

Example: This example shows how blob-slicing works with multiple threads.

Assumptions:

◆ POSIX pthread model

◆ Variables inStream1 and inStream2, based on type FPStreamRef, each of which holds half the data in a file, respectively.

- You have an open clip and an open tag into which to write the data (`vTag` is a `FPTagRef` type).
- Data structure used to pass data to a thread:

```
struct myWriterArgs {
   FPTagRef mTag;
   FPStreamRef mStream;
   FPLong mOptions;
   FPInt mSequenceID;
} myWriterArgs;
```

- Thread function:

```
void* myWriter (void* inArgs)
{
   myWriterArgs* vArgs = (myWriterArgs*)inArgs;
   if ( vArgs )
   {
      FPTag_BlobWritePartial(vArgs->mTag,
                                    vArgs->mStream,
                                    vArgs->mOptions,
                                    vArgs->mSequenceID);
   }
   return NULL;
}
```

Example of writing the data to a tag in an open C-Clip (`vTag`):

```
// initialize the data
myWriterArgs vArgs1;
myWriterArgs vArgs2;
vArgs1.mTag = vTag;
vArgs1.mStream = inStream1;
vArgs1.mOptions = FP_OPTION_CLIENT_CALCID;
vArgs1.mSequenceID = 1;
vArgs2.mTag = vTag;
vArgs2.mStream = inStream2;
vArgs2.mOptions = FP_OPTION_CLIENT_CALCID;
vArgs2.mSequenceID = 2;

// create the threads
pthread_t myThread1;
pthread_t myThread2;
pthread_create(&myThread1, NULL, &myWriter,
(void*)&vArgs1);
pthread_create(&myThread2, NULL, &myWriter,
(void*)&vArgs2);

// wait for the threads to complete
```

```
pthread_join(myThread1, NULL);
pthread_join(myThread2, NULL);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_ACK_NOT_RCV_ERR` (server error)
- `FP_BLOBIDFIELD_ERR` (server error)
- `FP_BLOBIDMISMATCH_ERR` (server error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_DUPLICATE_FILE_ERR` (internal error)
- `FP_DUPLICATE_ID_ERR` (client error)
- `FP_MULTI_BLOB_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FPParameterException` (program logic error or internal error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)
- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_SERVER_NO_CAPACITY_ERR` (server error)
- `FP_TAG_READONLY_ERR` (program logic error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_WRONG_STREAM_ERR` (client error)

Retention Class Functions

This section describes the following functions used to manage retention classes:

- FPRetentionClass_Close
- FPRetentionClassContext_Close
- FPRetentionClassContext_GetFirstClass
- FPRetentionClassContext_GetLastClass
- FPRetentionClassContext_GetNamedClass
- FPRetentionClassContext_GetNextClass
- FPRetentionClassContext_GetNumClasses
- FPRetentionClassContext_GetPreviousClass
- FPRetentionClass_GetName
- FPRetentionClass_GetPeriod

FPRetentionClass_Close

Syntax: `FPRetentionClass_Close (FPRetentionClassRef inClassRef)`

Return Value: `void`

Input Parameters: `FPRetentionClassRef inClassRef`

Concurrency Requirement: This function is thread safe.

Description: This function closes the given retention class and frees all related (memory) resources. Note that calling this function on a retention class that is already closed may produce unwanted results.

Parameters: `const FPRetentionClassRef inClassRef`
The reference to a retention class as returned by one of the `FPRetentionClassContext_Get`*`XXX`*`Class()` functions.

Example:
```
FPRetentionClass_Close(vRetentionClass);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClassContext_Close

Syntax:
```
FPRetentionClassContext_Close (FPRetentionClassContext
   inContextRef)
```

Return Value:
```
void
```

Input Parameters:
```
FPRetentionClassContext inContextRef
```

Concurrency Requirement:
This function is thread safe.

Description:
This function closes the given retention class context and frees all related (memory) resources. Note that calling this function on a retention class context that is already closed may produce unwanted results.

Parameters:
`FPRetentionClassContext inContextRef`
The reference to a retention class context object as returned from `FPPool_GetRetentionClassContext()`.

Example:
```
FPRetentionClassContext_Close (myRetentionClassContext);
```

Error Handling:
`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClassContext_GetFirstClass

Syntax:

```
FPRetentionClassContext_GetFirstClass (const
   FPRetentionClassContextRef inContextRef)
```

Return Value:

`FPRetentionClassRef`

Input Parameters:

`const FPRetentionClassContextRef inContextRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns a reference to the first retention class in the specified retention class context.

Note: The ordering of retention classes in the retention class context is undefined.

The following functions control the iterator for the retention class context:

- `FPRetentionClassContext_GetFirstClass()`
- `FPRetentionClassContext_GetNextClass()`
- `FPRetentionClassContext_GetPreviousClass()`
- `FPRetentionClassContext_GetLastClass()`

This function returns `NULL` if there are no classes in the retention class context.

Call `FPRetentionClass_Close()` when you are done with this object.

Parameters:

`const FPRetentionClassContextRef inContextRef`
The reference to a retention class context, as returned from the function `FPPool_GetRetentionClassContext()`.

Example:

```
myClass = FPRetentionClassContext_GetFirstClass(myClassC
   ontext);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClassContext_GetLastClass

Syntax:

```
FPRetentionClassContext_GetLastClass (const
   FPRetentionClassContextRef inContextRef)
```

Return Value:

`FPRetentionClassRef`

Input Parameters:

`const FPRetentionClassContextRef inContextRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns a reference to the last retention class in the specified retention class context.

Note: The ordering of retention classes in the retention class context is undefined.

The following functions control the iterator for the retention class context:

- `FPRetentionClassContext_GetFirstClass()`
- `FPRetentionClassContext_GetNextClass()`
- `FPRetentionClassContext_GetPreviousClass()`
- `FPRetentionClassContext_GetLastClass()`

Call `FPRetentionClass_Close()` when you are done with this object.

Parameters:

`const FPRetentionClassContextRef inContextRef`
The reference to a retention class context, as returned from the function `FPPool_GetRetentionClassContext()`.

Example:

```
myClass = FPRetentionClassContext_GetLastClass(myClassCo
   ntext);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClassContext_GetNamedClass

Syntax:

```
FPRetentionClassContext_GetNamedClass (const
   FPRetentionClassContextRef inContextRef, const char
   *inName)
```

Return Value:

```
FPRetentionClassRef
```

Input Parameters:

```
const FPRetentionClassContextRef inContextRef,
const char *inName
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function returns a reference to a retention class from a given retention class context by specifying the class name. This function returns `NULL` if the specified class name is not contained in the retention class context.

Call `FPRetentionClass_Close()` when you are done with this object.

Parameters:

- `const FPRetentionClassContextRef inContextRef`
  The reference to a retention class context, as returned from `FPPool_GetRetentionClassContext()`.
- `const char *inName`
  The name of the class to be retrieved.

Example:

```
myClass = FPRetentionClassContext_GetNamedClass(myClass
   Context, "Save_Email");
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)
- `FP_INVALID_NAME` (program logic error)

FPRetentionClassContext_GetNextClass

Syntax:

```
FPRetentionClassContext_GetNextClass (const
   FPRetentionClassContextRef inContextRef)
```

Return Value:

`FPRetentionClassRef`

Input Parameters:

`const FPRetentionClassContextRef inContextRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns a reference to the next retention class—the class following the class returned by the last iterator function call—in the specified retention class context. If no `getXClass()` is called previous to this function, this function returns the retention class at the first location. If there are no more retention classes to return, this function returns `NULL`.

Note: The ordering of retention classes in the retention class context is undefined.

The following functions control the iterator for the retention class context:

- `FPRetentionClassContext_GetFirstClass()`
- `FPRetentionClassContext_GetNextClass()`
- `FPRetentionClassContext_GetPreviousClass()`
- `FPRetentionClassContext_GetLastClass()`

Call `FPRetentionClass_Close()` when you are done with this object.

Parameters:

`const FPRetentionClassContextRef inContextRef`
The reference to a retention class context, as returned from the function `FPPool_GetRetentionClassContext()`.

Example:

```
myClass = FPRetentionClassContext_GetNextClass(myClassCo
   ntext);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClassContext_GetNumClasses

Syntax:

```
FPRetentionClassContext_GetNumClasses (const
   FPRetentionClassContextRef inContextRef)
```

Return Value:

`FPInt`

Input Parameters:

`const FPRetentionClassContextRef inContextRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns the number of retention classes defined in a retention class context. You must first open a retention class context for the pool using `FPPool_GetRetentionClassContext()`.

Parameters:

`const FPRetentionClassContextRef inContextRef`
The reference to a retention class context, as returned from `FPPool_GetRetentionClassContext()`.

Example:

```
myRetClassContext = FPPool_GetRetentionClassContext
   (myPool);
numClasses = FPRetentionClassContext_GetNumClasses
   (myRetClassContext);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClassContext_GetPreviousClass

Syntax:

```
FPRetentionClassContext_GetPreviousClass (const
   FPRetentionClassContextRef inContextRef)
```

Return Value:

`FPRetentionClassRef`

Input Parameters:

`const FPRetentionClassContextRef inContextRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns a reference to the previous retention class—the class preceding the class returned by the last iterator function call—in the specified retention class context. If no `getXClass()` is called previous to this function, this function returns the retention class at the last location. If there are no more retention classes to return, this function returns `NULL`.

Note: The ordering of retention classes in the retention class context is undefined.

The following functions control the iterator for the retention class context:

- `FPRetentionClassContext_GetFirstClass()`
- `FPRetentionClassContext_GetNextClass()`
- `FPRetentionClassContext_GetPreviousClass()`
- `FPRetentionClassContext_GetLastClass()`

Call `FPRetentionClass_Close()` when you are done with this object.

Parameters:

`const FPRetentionClassContextRef inContextRef`
The reference to a retention class context, as returned from the function `FPPool_GetRetentionClassContext()`.

Example:

```
myClass = FPRetentionClassContext_GetPreviousClass(myCla
   ssContext);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPRetentionClass_GetName

Syntax:

```
FPRetentionClass_GetName (const FPRetentionClassRef
   inClassRef, char *outName, FPInt *ioNameLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPRetentionClassRef inClassRef, FPInt *ioNameLen
```

Output Parameters

```
char *outName, FPInt *ioNameLen
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function retrieves the name of the given retention class. The name is returned in `outName`.

Call `FPRetentionClass_Close()` when you are done with this object.

Parameters:

- `const FPRetentionClassRef inClassRef`
  The reference to a retention class as returned by one of the `FPRetentionClassContext_Get`*`XXX`*`Class()` functions.
- `char *outName`
  The buffer that will store the name of the retention class. The name will be truncated if needed to the buffer length as specified by `ioNameLen`.
- `FPInt *ioNameLen`
  Input: The reserved length, in characters, of the `outName` buffer.
  Output: The actual length of the name, in characters, including the end-of-string character.

Example:

```
FPInt namesize = MAX_NAME_SIZE;
char name[MAX_NAME_SIZE];
FPRetentionClass_GetName (myRetClassContext, name,
   &namesize);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)

FPRetentionClass_GetPeriod

Syntax:

```
FPRetentionClass_GetPeriod (const FPRetentionClassRef
   inClassRef)
```

Return Value:

```
FPLong
```

Input Parameters:

```
const FPRetentionClassRef inClassRef
```

Concurrency Requirement:

This function is thread safe.

Description:

This function returns the retention period, in seconds, of the given retention class.

Parameters:

`const FPRetentionClassRef inClassRef`
The reference to a retention class as returned by one of the `FPRetentionClassContext_Get`*`XXX`*`Class()` functions.

Example:

```
vRetentionPeriod = FPRetentionClass_GetPeriod
   (myRetentionClass);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)

Stream Functions

Streams are generalized input/output channels similar to the HANDLE mechanism used in the Win32 API for files, the C++ iostream functionality, the C standard I/O FILE routines, or the Java InputStream/OutputStream facility.

All bulk data movement in the API is achieved by first associating a stream with a data sink or source (for example, a file, an in-memory buffer, or the standard output channel) and then performing operations on that stream.

The API stream functions implement streams generically. When creating a stream, the application developer has to use method pointers to indicate how subsequent stream functions should handle the stream. For more information on the available method pointers for stream functions, refer to *FPStream_CreateGenericStream* on page 1-199.

Stackable Stream Support

A stackable stream is a stream that calls another stream as part of its operation. An example is a stream that prepends bytes to another stream, or a stream that compresses stream data. To support stackable streams, the following functions are available to the API: `FPStream_PrepareBuffer()`, `FPStream_Complete()`, `FPStream_SetMark()`, and `FPStream_ResetMark()`.

Generic Stream Operation

In versions of the SDK prior to v1.2, generic streams that were used for output (for example with `FPTag_BlobRead()`) received a buffer with data which the completion callback function had to process. For SDK v1.2 and higher, the application has the option of providing a buffer to the output stream that the SDK will fill. In this case, the stream remains the owner of the output buffer.

Foreign Pointer Mode

The SDK remains the owner of the pointer to the data buffer (`mBuffer`). The stream completion function is only allowed to read data from it. This mode is compatible with pre 1.2 SDK releases.

The following rules apply:

- Do not declare `prepareBufferProc`. If declared, `mBuffer` should be set to `NULL`.
- This function `completeProc` can read the data from `mBuffer` for `mTransferLen` bytes but it is not allowed to change these fields in the StreamInfo structure.

If the end of the stream has been reached, then `mAtEOF` is true. It is possible that this last call to the `completeProc` does not pass any data (`mTransferLen` is 0, and `mBuffer` remains `NULL`).

Stream Buffer Mode

The stream provides a buffer that the SDK will fill. `prepareBufferProc` is responsible for preparing this buffer and to set `mBuffer` to it. `mTransferLen` contains the number of bytes that can be transferred. `completeProc` processes this buffer. `mTransferLen` then contains the number of bytes actually transferred into the buffer. The value of `mBuffer` does not change.

The following rules apply:

- `prepareBufferProc` should set `mBuffer` to point to the buffer that it manages and it should set `mTransferLen` to the maximum number of bytes that the stream can receive (typically `mTransferLen` is the length of `mBuffer`).
- `completeProc` can process the data in `mBuffer`. `mTransferLen` number of bytes are actually copied into `mBuffer`.

The algorithm is as follows:

set `vRemainInBuffer` to 0
set `vRemainInPacket` to 0
loop until all blob data has been read
  if `vRemainInBuffer` == 0 then
    `FPStream_PrepareBuffer`
    set `vRemainInBuffer` to `mTransferLen`
  if `vRemainInPacket` == 0 then
    `ReadPacket` from Centera
    set `vRemainInPacket` to length of data
  set `vToCopy` to min (`vRemainInBuffer`, `vRemainInPacket`)
  copy `vToCopy` bytes from packet to `mBuffer`
  decrease `vRemainInPacket` by `vToCopy`
  decrease `vRemainInBuffer` by `vToCopy`
  if `vRemainInBuffer` == 0 then
    `FPStreamComplete`

If the end of the stream has been reached, then `mAtEOF` is true. It is possible that this last call to `completeProc` does not pass any data (`mTransferLen` is 0).

Stream Creation Functions

This section describes the following functions used to create a stream:

- `FPStream_CreateBufferForInput`
- `FPStream_CreateBufferForOutput`
- `FPStream_CreateFileForInput`
- `FPStream_CreateFileForOutput`
- `FPStream_CreateGenericStream`
- `FPStream_CreateTemporaryFile`
- `FPStream_CreateToNull`
- `FPStream_CreateToStdio`

FPStream_CreateBufferForInput

Syntax:

```
FPStream_CreateBufferForInput (char *inBuffer, const
   unsigned long inBuffLen)
```

Return Value:

`FPStreamRef`

Input Parameters:

`char *inBuffer, const unsigned long inBuffLen`

Concurrency Requirement:

This function is thread safe.

Description:

This function creates a stream to read from a memory buffer and returns a reference to the created stream.

Parameters:

- `const char *inBuffer`
  `inBuffer` is the memory buffer containing the data for the stream.
- `const unsigned long inBuffLen`
  `inBuffLen` is the buffer length (in bytes) of `inBuffer`.

Example:

```
char myDataSource[A_BIG_SIZE];
myStream = FPStream_CreateBufferForInput (myDataSource,
   A_BIG_SIZE);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)

FPStream_CreateBufferForOutput

Syntax:

```
FPStream_CreateBufferForOutput (char *inBuffer, const
   unsigned long inBuffLen)
```

Return Value:

`FPStreamRef`

Input Parameters:

`const char *inBuffer, const unsigned long inBuffLen`

Concurrency Requirement:

This function is not thread safe.

Description:

This function creates a stream to write to a memory buffer and returns a reference to the created stream. When the end of the buffer has been reached `mAtEOF` of `pStreamInfo` is set to true. Refer to *FPStream_CreateGenericStream* on page 1-199 for more information on `pStreamInfo`.

Parameters:

- `const char *inBuffer`
  `inBuffer` is the memory buffer containing the data for the stream.
- `const unsigned long inBuffLen`
  `inBuffLen` is the buffer length (in bytes) of `inBuffer`.

Example:

```
char myDataSource[A_BIG_SIZE];
myStream = FPStream_CreateBufferForOutput (myDataSource,
   A_BIG_SIZE);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)

FPStream_CreateFileForInput

Syntax: `FPStream_CreateFileForInput (const char *inFilePath, const char *inPerm, const long inBuffSize)`

Return Value: `FPStreamRef`

Input Parameters: `const char *inFilePath, const char *inPerm, const long inBuffSize`

Concurrency Requirement: This function is thread safe.

Description: This function creates a stream to read from a file and returns a reference to the created stream. The stream behaves like a file and depending on the given permission (`inPerm`), you can use the stream with most of the stream handling functions (refer to Page 1-215 for more information).

Parameters:

- `const char *inFilePath`
  `inFilePath` is the buffer that contains the path name of the file for which the stream must be created.
- `const char *inPerm`
  `inPerm` is the buffer that contains the open permission for the file. You can use the following permission:

  rb: opens the given file for reading. If the file does not exist or the system cannot find the file, the function returns an error.
- `const long inBuffSize`
  `inBuffSize` is the size of the buffer (in bytes) that is used when reading the file. The value has to be greater than 0.

Example: This example shows how to stream data to a pool.

```
  { FPStreamRef vStream = FPStream_CreateFileForInput (pPath, "rb", 16*1024);
// open a new stream
    if (vStream != 0)
      { FPTag_BlobWrite (vFileTag, vStream, 0);
// write it to the pool
        FPStream_Close (vStream);
// and don't forget to close it...
      }
    vStatus = FPPool_GetLastError();
  }
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_FILESYS_ERR` (program logic error)

FPStream_CreateFileForOutput

Syntax:

```
FPStream_CreateFileForOutput (const char *inFilePath,
   const char *inPerm)
```

Return Value:

```
FPStreamRef
```

Input Parameters:

```
const char *inFilePath, const char *inPerm
```

Concurrency Requirement:

This function is not thread safe.

Description:

This function creates a stream to write to a file and returns a reference to the created stream. The stream behaves like a file and depending on the given permission (`inPerm`), you can use the stream with all stream handling functions (refer to *Stream Handling Functions* on page 1-215 for more information).

Parameters:

- `const char *inFilePath`
  `inFilePath` is the buffer that contains the pathname of the file for which the stream must be created.
- `const char *inPerm`
  `inPerm` is the buffer that contains the open permission for the file, for writing this is usually **wb**. You can use one of the following permissions:

  wb: opens the given file for writing. This function overwrites existing content of the file.

  ab: opens the file for writing at the end of the file. If the file does not exist, the function creates the file.

  rb+: opens the given file for both reading and writing. If the file does not exist or the system cannot find the file, the function returns an error.

  wb+: opens the given file for both reading and writing. If the given file exists, the function overwrites the content.

  ab+: opens the given file for reading and writing at the end of the file. If the file does not exist, the function creates the file.

FPStream_CreateGenericStream

Syntax:

```
FPStream_CreateGenericStream (const FPStreamProc
   inPrepareBufferProc, const FPStreamProc
   inCompleteProc, const FPStreamProc inSetMarkerProc,
   const FPStreamProc inResetMarkerProc, const
   FPStreamProc inCloseProc, const void *inUserData)
```

Return Value: `FPStreamRef`

Input Parameters:

```
const FPStreamProc inPrepareBufferProc, const
FPStreamProc inCompleteProc, const FPStreamProc
inSetMarkerProc, const FPStreamProc inResetMarkerProc,
const FPStreamProc inCloseProc, const void *inUserData
```

Concurrency Requirement: This function is thread safe.

Description: This function allocates a stream data structure, declares its methods and returns a reference to the created stream. This function returns `NULL` if the stream has not been created.

If you want to extend the created Generic Stream, you must supply one or more function pointers.

Parameters:

`const void *inUserData`
Any data that the application wants to pass to the method pointers.

The callback functions are of type `FPStreamProc`:

```
typedef long (*FPStreamProc) (FPStreamInfo*)
```

They take a pointer to `FPStreamInfo` as parameter and return an error if unsuccessful or `ENOERR` if successful.

Note: The application must ensure that the callback functions are thread safe and take no more than 1 minute to execute.

In the following callback function descriptions, **input stream** refers to a stream from which the SDK has to read data, for example when writing a blob to the cluster. **Output stream** refers to a stream to which the SDK has to write data, for example when reading a blob from the cluster.

FPStreamProc inPrepareBufferProc
This method prepares a buffer that the stream can use. If the stream is an input buffer, the callback method prepares a buffer that contains the data. The `mBuffer` field of `FPStreamInfo` contains a pointer to the data. If the stream is an output buffer, the function does nothing.

If you name your function `myPrepareBuffer` then you would declare it like this:

```
static long myPrepareBuffer (FPStreamInfo *pStreamInfo)
```

`pStreamInfo` is a pointer to an `FPStreamInfo` structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 1-7, *FPStreamInfo fields for myPrepareBuffer*, for a description of all possible fields of `FPStreamInfo`.

Table 1-7 FPStreamInfo fields for myPrepareBuffer

Field	Type	Description
mVersion	short	Version of the `FPStreamInfo` structure. Currently the value of this field is 3. Check this field for backward compatibility.
mUserData	void*	Parameter passed unchanged from `FPStream_CreateGenericStream` to each callback function. You can use this to pass (a pointer to) myStream-specific data to the callback instead of relying on global variables.
mStreamPos	FPLong	The current position in the stream where input occurs. This field should be updated by `myPrepareBuffer` if needed.
mStreamLen	FPLong	The total length of the stream. This is –1 (unknown) by default and should be updated by `myPrepareBuffer`. It must be initialized before the first call to `prepareBuffer` (before `FPTag_BlobWrite` is called).
mAtEOF	FPBool	Indicates if `myStream` has reached the end of stream or not. `myPrepareBuffer` should return true in this field if the last segment is read.

Table 1-7 FPStreamInfo fields for myPrepareBuffer (continued)

Field	Type	Description
mReadFlag	FPBool	Indicates if `myStream` is used for reading (input) or writing (output). The behavior of `myPrepareBuffer` might depend on the value of this field. **In case of an input stream:** Must be set to true. **In case of an output stream:** Must be set to false.
mBuffer	void*	**In case of an input stream:** `myPrepareBuffer` should make `mBuffer` point to a buffer containing data. Possibly `myPrepareBuffer` must allocate memory, read data from a device into that memory, and set `mBuffer` to point to it. **In case of an output stream:** This field should either be set to a stream-managed buffer or it should be set to NULL.
mTransferLen	FPLong	**In case of an input stream:** `mTransferLen` indicates the maximum number of bytes that the SDK wants to receive from the input. `myPrepareBuffer` returns the actual number of bytes in `mBuffer` in this field. If the buffer contains no data, then mTransferLen = 0. If mTransferLen = 0 and `mAtEOF` is true, then the end of the data stream has been reached. **In case of an output stream:** If `myPrepareBuffer` manages the output buffer, then `mTransferLen` should be set to the size of that output buffer. The SDK transfers the number of bytes into the output buffer that equals its size. If the stream does not manage the output buffer, then this field is unused.

The SDK calls `myPrepareBuffer` prior to transferring data from the input stream to the Centera server. The SDK expects the callback function to make the data available and provides a pointer to it in `mBuffer`. The pointer to this data is 'owned' by the SDK until the `completeProc` is called. The `prepareBufferProc` is also called before data is transferred from the Centera server to the SDK. `myPrepareBuffer` could then be used to prepare the output stream to receive data. In many cases however, it is not necessary to implement this function for output streams (pass `NULL` in `FPStream_CreateGenericStream()`). If used for output streams,

myPrepareBuffer can put a pointer to the buffer that it manages in mBuffer and its length in mTransferLen.

FPStreamProc inCompleteProc
The generic stream calls this method when the buffer that has been prepared with inPrepareBufferProc is no longer needed. If the stream was an input stream, this means the data has been processed successfully. If the stream was an output stream, this means the buffer contains the requested data and it can be written to an output device.

If you name your function myComplete then you would declare it like this:

```
static long myComplete (FPStreamInfo *pStreamInfo)
```

pStreamInfo is a pointer to an FPStreamInfo structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 1-8, *FPStreamInfo fields for myComplete*, for a description of all possible fields of FPStreamInfo.

Table 1-8 FPStreamInfo fields for myComplete

Field	Type	Description
mVersion	short	Version of the FPStreamInfo structure. Currently the value of this field is 3. Check this field for backward compatibility.
mUserData	void*	Parameter passed unchanged from FPStream_CreateGenericStream to each callback function. You can use this to pass (a pointer to) myStream-specific data to the callback instead of relying on global variables.
mStreamPos	FPLong	The current position in the stream where input or output occurs. This field should be updated by myComplete if needed. This should preferably already have been initialized when myStream is created.
mStreamLen	FPLong	The total length of the stream. This is –1 (unknown) by default and should be updated by myComplete. It must be initialized before the first call to prepareBuffer (before FPTag_BlobWrite is called).
mAtEOF	FPBool	**In case of an output stream:** Indicates that the last buffer of data will now be written. If mTransferLen = 0, then no data will actually be passed.

Table 1-8 FPStreamInfo fields for myComplete (continued)

Field	Type	Description
mReadFlag	FPBool	Indicates if `myStream` is used for reading (input) or writing (output). This field is set by the SDK and should not be changed by the callback function. The behavior of `myComplete` might depend on the value of this field.
mBuffer	void*	**In case of an input stream:** Contains a pointer to the data that has been read (as provided by an earlier call from `myPrepareBuffer`). **In case of an output stream:** Contains the address of a memory buffer that holds the data that has to be written. If this pointer is owned by the SDK (refer to `prepareBufferProc`) then the callback function is allowed to access this memory during the execution of `myComplete` only.
mTransferLen	FPLong	**In case of an input stream:** Contains the number of bytes that have been read (as provided by an earlier call from `myPrepareBuffer`). **In case of an output stream:** Contains the number of bytes that `mBuffer` points to. This value can be 0 (in which case `mAtEOF` is true).

In case of an output stream, `myComplete` actually transfers the data pointed to by `mBuffer` to the output device. In case of a file, this might translate into an `fwrite()` operation. The `mBuffer` pointer can either be provided by the stream or by the SDK.

Note: If the SDK provides the `mBuffer` pointer to the stream to read the output data, the callback function should never change this data.

In case of an input stream, the callback function notifies that the SDK has finished with the input buffer (`mBuffer`). It can then unlock or deallocate the buffer if necessary. In many cases, this callback is `NULL` for input streams.

In case of an output stream, the end of the stream behaves as follows:

- If the application allocated its buffer in `prepareBufferProc`, then the last `completeProc` callback has `mTransferLen` >= 0 and `mAtEOF` is true.
- If the application did not allocate its buffer, then the last call to `completeProc` always has `mTransferLen` = 0 and `mAtEOF` is true.

FPStreamProc inSetMarkerProc
This method instructs the generic stream to mark the current position in the stream. If the stream supports marking, the function can use the `mMarkerPos` field to indicate the current position.

If you name your function `mySetMarker` then you would declare it like this:

```
static long mySetMarker (FPStreamInfo *pStreamInfo)
```

`pStreamInfo` is a pointer to an `FPStreamInfo` structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 1-9, *FPStreamInfo fields for mySetMarker*, for a description of all possible fields of `FPStreamInfo`.

Table 1-9 FPStreamInfo fields for mySetMarker

Field	Type	Description
mVersion	short	Version of the `FPStreamInfo` structure. Currently the value of this field is 3. Check this field for backward compatibility.
mUserData	void*	Parameter passed unchanged from `FPStream_CreateGenericStream` to each callback function. You can use this to pass (a pointer to) myStream-specific data to the callback instead of relying on global variables.
mStreamPos	FPLong	The current position in the stream where input or output occurs. This field should be updated by `mySetMarker` to reflect the current position in the stream. In case of a file-based stream, this is often an `ftell()` call.
mReadFlag	FPBool	Indicates if `myStream` is used for reading (input) or writing (output). This field is set by the SDK and should not be changed by the callback function. This callback function usually does not use this field.
mMarkerPos	FPLong	This field is updated by GenericSreams to reflect `mStreamPos` after the callback function returns.

The SDK sometimes needs to return to an earlier position in the stream. To do this, it sets a 'mark' at the current position in the stream (how the current position is defined depends on the stream implementation). If necessary, the SDK can return later to that position by calling the `resetMarkerProc`.

If a stream does not support marking, then pass NULL in FPStream_CreateGenericStream.

Note: This callback function returns an error if unsuccessful or ENOERR if successful. Any error from mySetMarker is currently ignored by the SDK as it is not a fatal condition.

FPStreamProc inResetMarkerProc
This method tells the stream to go back to the marked position in the stream (mMarkerPos).

If you name your function myResetMarker then you would declare it like this:

```
static long myResetMarker (FPStreamInfo *pStreamInfo)
```

pStreamInfo is a pointer to an FPStreamInfo structure. This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: input or output. Refer to Table 1-10, *FPStreamInfo fields for myResetMarker*, for a description of all possible fields of FPStreamInfo.

Table 1-10 FPStreamInfo fields for myResetMarker

Field	Type	Description
mVersion	short	Version of the FPStreamInfo structure. Currently the value of this field is 3. Check this field for backward compatibility.
mUserData	void*	Parameter passed unchanged from FPStream_CreateGenericStream to each callback function. You can use this to pass (a pointer to) myStream-specific data to the callback instead of relying on global variables.

Table 1-10 FPStreamInfo fields for myResetMarker (continued)

Field	Type	Description
mStreamPos	FPLong	The current position in the stream where input or output occurs. This field is updated by GenericStreams to the value of `mMarkerPos` after the call to `myResetMarker`.
mReadFlag	FPBool	Indicates if `myStream` is used for reading (input) or writing (output). This field is set by the SDK and should not be changed by the callback function. This callback function does not usually use this field.
mMarkerPos	FPLong	If the stream implementation uses an offset field to remember the marked position, then `mMarkerPos` contains the position of the mark. `myResetMarker` should set the stream to that position. In case of file-based streams, `myResetMarker` should do a `fseek()` on `mMarkerPos`.

When the SDK needs to return to an earlier position in the stream (indicated by calling `setMarkerProc`), it calls `resetMarkerProc`.

If a stream does not support marking, then pass `NULL` in `FPStream_CreateGenericStream`. When the SDK needs this functionality, it exits with an `FP_OPERATION_REQUIRES_MARK` error.

In some cases, the stream cannot return to an arbitrary position in the stream, but can only go back to the beginning of it (for example if the data needs to be calculated). If the callback returns `FP_OPERATION_REQUIRES_MARK` in the first case, the SDK retries with `mMarkerPos 0`. The stream can then reset itself and the SDK retries the complete operation.

`FPStreamProc inCloseProc`
This method informs that the application has performed its operations on the stream and that the stream can clean up the resources that it has allocated.

If you name your function `myClose` then you would declare it like this:

```
static long myClose (FPStreamInfo *pStreamInfo)
```

This structure is allocated and maintained by the Generic Stream. You are allowed to read and write fields from this structure. The exact meaning of each field depends on the purpose of the stream: for input

or for output. Refer to Table 1-11, *FPStreamInfo fields for myClose*, for a description of all possible fields of `FPStreamInfo`.

Table 1-11 FPStreamInfo fields for myClose

Field	Type	Description
mVersion	short	Version of the `FPStreamInfo` structure. Currently the value of this field is 3. Check this field for backward compatibility.
mUserData	void*	Parameter passed unchanged from `FPStream_CreateGenericStream` to each callback function. You can use this to pass (a pointer to) `myStream`-specific data to the callback instead of relying on global variables.

When the SDK has finished its operations on a stream, it calls the `closeProc` callback function. `myClose` then has the chance to close any opened resources (for example a file or a network socket) and to deallocate any memory if necessary. If no close function is needed, then `NULL` can be passed to `FPStream_CreateGenericStream`.

Example: The sample code shows how to build a stream implementation on top of Generic Streams. Error handling, parameter checking and special cases are omitted for clarity.

File Input

This example creates a File Input stream. The parameters are a path to the file, the file permissions, and the buffersize. A memory buffer is allocated to read the file data. This buffer is deallocated in the close method. The `TStreamFileInfo` is a structure holding some data specific for file streams. A pointer to it is passed along using the `mUserData` field.

```
EXPORT FPStreamRef FPStream_CreateFileForInput (const char *inFilePath, const
   char *inPerm, const long inBuffSize)
{ TStreamFileInfo *vFileInfo = NULL;
  char            *vBuffer = NULL;
  FPStreamRef      vResult;

  vFileInfo = (TStreamFileInfo*) calloc (1, sizeof (TStreamFileInfo));

  vBuffer    = (char*) malloc (inBuffSize);
  vFileInfo->mBufferLen = inBuffSize;

  // build the access rights from the inPerm parameter here ..
```

```
    vFileInfo->mFile = CreateFile (inFilePath, access, FILE_SHARE_READ, NULL,
   createMode, FILE_ATTRIBUTE_NORMAL, NULL);

  vResult = FPStream_CreateGenericStream (fileInPrepBuffer, NULL,
                                          fileSetMarker, fileResetMarker,
                                          fileClose, vFileInfo);

  // set some FPStreamInfo fields
  FPStreamInfo *vInfo = FPStream_GetInfo (vResult);
  if (vInfo)
    { vInfo->mReadFlag = true; // indicate it's an input stream
      vInfo->mBuffer   = vBuffer; // set buffer
      vInfo->mStreamLen = // get the file length;
    }

  return vResult;
}
```

A part of the file is read into the **buffer**.

```
static long fileInPrepBuffer (FPStreamInfo *pStreamInfo)
{ TStreamFileInfo *vFileInfo = (TStreamFileInfo*) pStreamInfo->mUserData;
  unsigned long    l;

  if (ReadFile (vFileInfo->mFile, pStreamInfo->mBuffer,
                          vFileInfo->mBufferLen, &l, NULL) == 0)
    return GetLastError ();

  pStreamInfo->mTransferLen = l;
  pStreamInfo->mStreamPos += l;
  if (l < vFileInfo->mBufferLen)
    pStreamInfo->mAtEOF = true;

  return ENOERR;
}
```

The **marker** methods use a type of 'seek()' function to set and get the current position in the file that is read or written. The offset is kept in the `mMarker` field (which is only used by stream handling functions and will not be changed by Generic Streams).

```
static long fileSetMarker (FPStreamInfo *pStreamInfo)
{ TStreamFileInfo *vFileInfo = (TStreamFileInfo*) pStreamInfo->mUserData;

  pStreamInfo->mMarkerPos = myFileSeek (vFileInfo->mFile, 0, FILE_CURRENT);

  return ENOERR;
}

static long fileResetMarker (FPStreamInfo *pStreamInfo)
{ TStreamFileInfo *vFileInfo = (TStreamFileInfo*) pStreamInfo->mUserData;

  myFileSeek (vFileInfo->mFile, pStreamInfo->mMarkerPos, FILE_BEGIN);
  pStreamInfo->mStreamPos = pStreamInfo->mMarkerPos;
```

```
  return ENOERR;
}
```

This method **deallocates the buffer** (allocated in the stream creation function) and closes the file.

```
static long fileClose (FPStreamInfo *pStreamInfo)
{ TStreamFileInfo *vFileInfo = (TStreamFileInfo*) pStreamInfo->mUserData;

  CloseHandle (vFileInfo->mFile);

  if (vFileInfo->mBufferLen > 0)
    free (pStreamInfo->mBuffer);

  free (vFileInfo);

  return ENOERR;
}
```

File Output

This method is very similar to the stream creation function for File Input. The main difference is that no memory buffer is allocated.

```
EXPORT FPStreamRef FPStream_CreateFileForOutput (const char *inFilePath, const
   char *inPerm)
{ TStreamFileInfo *vFileInfo = NULL;
  FPStreamRef      vResult;

  vFileInfo = (TStreamFileInfo*) calloc (1, sizeof (TStreamFileInfo));

  // get the access rights from the inPerm parameter..

    vFileInfo->mFile = CreateFile (inFilePath, access, FILE_SHARE_READ, NULL,
   createMode, FILE_ATTRIBUTE_NORMAL, NULL);

  vResult = FPStream_CreateGenericStream (NULL, fileOutComplete,
                                          fileSetMarker, fileResetMarker,
                                          fileClose, vFileInfo);

  FPStreamInfo *vInfo = FPStream_GetInfo (vResult);
  if (vInfo)
    { vInfo->mReadFlag = false; // stream is for output
      vInfo->mStreamLen = // get file length (usually = 0)
    }

  return vResult;
}
```

This method **writes** the data pointer to `mBuffer` into a file. The call back function returns an error value to the application to stop `FPTag_BlobRead` from the stream.

```
static long fileOutComplete (FPStreamInfo *pStreamInfo)
```

```
{ TStreamFileInfo *vFileInfo = (TStreamFileInfo*) pStreamInfo->mUserData;
  unsigned long    l;

  if (WriteFile (vFileInfo->mFile, pStreamInfo->mBuffer,
                 pStreamInfo->mTransferLen, &l, NULL) == 0)
    { return FP_OPERATION_NOT_SUPPORTED[1];
    }
  pStreamInfo->mTransferLen = l;
  pStreamInfo->mStreamPos  += l;

  return ENOERR;
}
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful.

Example: This example shows how to write data from a pool to a stream.

```
{
FPStreamRef vStream = FPStream_CreateFileForOutput (pPath, "wb");
// create a new binary file for write
    if (vStream == 0)
      return FPPool_GetLastError();
    FPTag_BlobRead (inTag, vStream, 0);
// read stream has highest performance for downloading
    FPStream_Close (vStream);
// close file stream
}
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_FILESYS_ERR` (program logic error)

1. Or any other error value. The error value will be returned to the application.

FPStream_CreateTemporaryFile

Syntax: `FPStream_CreateTemporaryFile (const long inMemBuffSize)`

Return Value: `FPStreamRef`

Input Parameters: `const long inMemBuffSize`

Concurrency Requirement: This function is thread safe.

Description: This function creates a stream for temporary storage and returns a reference to the created stream. If the length of the stream exceeds `inMemBuffSize`, the overflow is flushed to a temporary file in the platform-specific temporary directory. This temporary file is automatically deleted when the stream closes.

Parameters: `const long inMemBuffSize`
`inMemBuffSize` is the size of the memory buffer.

Example:
```
myStream = FPStream_CreateTemporaryFile(2048);
 * Use a 2048 byte in memory buffer for the file.
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)

FPStream_CreateToNull

Syntax: `FPStream_CreateToNull (void)`

Return Value: `FPStreamRef`

Concurrency Requirement: This function is thread safe.

Description: This function creates a stream for output but does not write the bytes. This function returns a reference to the created stream.

Parameters: `void`

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful.

FPStream_CreateToStdio

Syntax: `FPStream_CreateToStdio (void)`

Return Value: `FPStreamRef`

Concurrency Requirement: This function is thread safe.

Description: This function creates a stream for output to the console. The stream can be used only for writing. This function returns a reference to the created stream.

Parameters: `void`

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful.

Stream Handling Functions

This section describes the following functions that handle a stream:

- `FPStream_Close`
- `FPStream_Complete`
- `FPStream_GetInfo`
- `FPStream_PrepareBuffer`
- `FPStream_ResetMark`
- `FPStream_SetMark`

FPStream_Close

Syntax: `FPStream_Close (const FPStreamRef inStream)`

Return Value: `void`

Input Parameters: `const FPStreamRef inStream`

Concurrency Requirement: This function is thread safe.

Description: This function closes the given stream. Note that calling this function on a stream that has already been closed may produce unwanted results.

Note: Always use this function to close streams that are no longer needed in order to prevent memory leaks.

Parameters: `const FPStreamRef inStream`
The reference to a stream (as returned from the functions `FPStream_CreateXXX()` or `FPStream_CreateGenericStream()`).

Example:
```
FPStream_Close (myStream);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPStream_Complete

Syntax: `FPStream_Complete (const FPStreamRef inStream)`

Return Value: `FPStreamInfo*`

Input Parameters: `const FPStreamRef inStream`

Concurrency Requirement: This function is thread safe if the callback function is thread safe.

Description: This function calls `completeProc` from the stream. `completeProc` was previously passed to `FPStream_CreateGenericStream`. If no `completeProc` callback is defined for this stream, then the function does nothing.

This function returns a pointer to the StreamInfo structure. This pointer is identical to the one that is returned by `FPStream_GetInfo`.

Parameters: `const FPStreamRef inStream`
The reference to a stream as created by `FPStream_CreateGenericStream()`.

Example: Refer to the *Centera Programmer's Guide*, P/N 069001127, for information about code examples provided with the SDK package.

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- Any error that is returned by the `completeProc` callback function.

FPStream_GetInfo

Syntax: `FPStream_GetInfo (const FPStreamRef inStream)`

Return Value: `FPStreamInfo*`

Input Parameters: `const FPStreamRef inStream`

Concurrency Requirement: This function is thread safe.

Description: This function returns information about the given stream.

Parameters: `const FPStreamRef inStream`
The reference to a stream.

Example: Refer to the example section of *FPStream_CreateGenericStream* on page 1-199 for an example of `FPStream_GetInfo()`.

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPStream_PrepareBuffer

Syntax: `FPStream_PrepareBuffer (const FPStreamRef inStream)`

Return Value: `FPStreamInfo*`

Input Parameters: `const FPStreamRef inStream`

Concurrency Requirement: This function is thread safe if the callback function is thread safe.

Description: This function sets `mBuffer` and `mTransferLen` for an output stream to `NULL`. The SDK can thus detect that the application has provided a buffer. If `mBuffer` is `NULL` when the SDK wants to write data, it provides a pointer to its own buffer.

This function then calls the `prepareBufferProc` from the stream. `prepareBufferProc` was previously passed to `FPStream_CreateGenericStream`. If no `prepareBufferProc` callback is defined for this stream, then the function does nothing.

This function returns a pointer to the StreamInfo structure. This pointer is identical to the one that is returned by `FPStream_GetInfo`.

Parameters: `const FPStreamRef inStream`
The reference to a stream as created by `FPStream_CreateGenericStream()`.

Example: Refer to the *Centera Programmer's Guide*, P/N 069001127, for information about code examples provided with the SDK package.

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- Any error that is returned by the `prepareBufferProc` callback function.

FPStream_ResetMark

Syntax: `FPStream_ResetMark (const FPStreamRef inStream)`

Return Value: `void`

Input Parameters: `const FPStreamRef inStream`

Concurrency Requirement: This function is thread safe if the callback function is thread safe.

Description: This function calls `resetMarkerProc` from the stream. `resetMarkerProc` was previously passed to `FPStream_CreateGenericStream`. If no `resetMarkerProc` callback is defined for this stream, then the function returns the error `FP_OPERATION_REQUIRES_MARK` unless the current position in the stream equals the marked position (the fields `mMarkerPos` and `mStreamPos` in `FPStreamInfo` are equal).

Parameters: `const FPStreamRef inStream`
The reference to a stream as created by `FPStream_CreateGenericStream()`.

Example: Refer to the *Centera Programmer's Guide*, P/N 069001127, for information about code examples provided with the SDK package.

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OPERATION_REQUIRES_MARK` (program logic error)
- Any error that is returned by the `setMarkerProc` callback function.

FPStream_SetMark

Syntax: `FPStream_SetMark (const FPStreamRef inStream)`

Return Value: `void`

Input Parameters: `const FPStreamRef inStream`

Concurrency Requirement: This function is thread safe if the callback function is thread safe.

Description: This function calls `setMarkerProc` from the stream. `setMarkerProc` was previously passed to `FPStream_CreateGenericStream`. If no `setMarkerProc` callback is defined for this stream, then the function returns the error `FP_OPERATION_REQUIRES_MARK`. The application usually ignores this error.

Parameters: `const FPStreamRef inStream`
The reference to a stream as created by `FPStream_CreateGenericStream()`.

Example: Refer to the *Centera Programmer's Guide*, P/N 069001127, for information about code examples provided with the SDK package.

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OPERATION_REQUIRES_MARK` (program logic error)
- Any error that is returned by the `setMarkerProc` callback function.

Query Functions

The Access API provides functions that query C-Clips—both existing and deleted (*reflections*)—stored on a Centera cluster.

The query feature is intended for backup applications and not as a general-purpose application feature. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on the query feature.

This section describes the following query functions:

Query Expression Functions

- `FPQueryExpression_Close`
- `FPQueryExpression_Create`
- `FPQueryExpression_DeselectField`
- `FPQueryExpression_GetEndTime`
- `FPQueryExpression_GetStartTime`
- `FPQueryExpression_GetType`
- `FPQueryExpression_IsFieldSelected`
- `FPQueryExpression_SelectField`
- `FPQueryExpression_SetEndTime`
- `FPQueryExpression_SetStartTime`
- `FPQueryExpression_SetType`

Pool Query Functions

- `FPPoolQuery_Close`
- `FPPoolQuery_FetchResult`
- `FPPoolQuery_GetPoolRef`
- `FPPoolQuery_Open`

Query Result Functions

- `FPQueryResult_Close`
- `FPQueryResult_GetClipID`
- `FPQueryResult_GetField`
- `FPQueryResult_GetResultCode`
- `FPQueryResult_GetTimestamp`
- `FPQueryResult_GetType`

FPQueryExpression_Close

Syntax:

```
FPQueryExpression_Close (const FPQueryExpressionRef
   inRef)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryExpressionRef inRef
```

Concurrency Requirement:

This function is thread safe.

Description:

This function closes the query expression and releases all allocated resources. Call this function when your application no longer needs the `FPQueryExpressionRef` object. Note that calling this function on a query expression that has already been closed may produce unwanted results.

Parameters:

`const FPQueryExpressionRef inRef`
The reference to a query expression as returned from `FPQueryExpression_Create()`.

Example:

```
FPQueryExpression_Close (myQueryExp);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPQueryExpression_Create

Syntax: `FPQueryExpression_Create (void)`

Return Value: `FPQueryExpressionRef`

Concurrency Requirement: This function is thread safe.

Description: This function creates a query expression, which defines the conditions used to query the pool. You must call `FPQueryExpression_Create()` before calling `FPPoolQuery_Open()`.

You can modify the conditions for the query by calling `FPQueryExpression_SetStartTime()`, `FPQueryExpression_SetEndTime()`, and `FPQueryExpression_SetType()`. By default, the query expression queries all existing C-Clips (start time = 0, end time = -1, type = `FP_QUERY_TYPE_EXISTING`).

You specify what description attributes, if any, you want included in the query results by calling `FPQueryExpression_SelectField()`. By default, the query returns no description attributes, so the application only has access to the Clip ID of returned C-Clips.

When your query is complete, call `FPQueryExpression_Close()` to free any allocated resources.

Parameters: `void`

Example:

```
FPQueryExpressionRef myQueryExp = 0;

myQueryExp = FPQueryExpresion_Create();
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_OUT_OF_MEMORY_ERR` (client error)

FPQueryExpression_DeselectField

Syntax:

```
FPQueryExpression_DeselectField (const
   FPQueryExpressionRef inRef, const char *inAttrName)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryExpressionRef inRef, const char *inAttrName
```

Concurrency Requirement:

This function is thread safe.

Description:

This function removes a previously selected description attribute from being included in the query result. No error is returned if the query expression does not contain the specified attribute; that is, the attribute was not previously selected using `FPQueryExpression_SelectField()`.

Refer to *FPQueryExpression_SelectField* on page 1-231 for information on description attributes.

Parameters:

- `const FPQueryExpressionRef inRef`
  The reference to a query expression as returned from `FPQueryExpression_Create()`.
- `const char *inAttrName`
  The name of a description attribute.

Example:

```
char* Company = "Company XYZ";

FPQueryExpression_DeselectField (myQuery, Company);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_GetEndTime

Syntax:
```
FPQueryExpression_GetEndTime (const FPQueryExpressionRef
   inRef)
```

Return Value:
`FPLong`

Input Parameters:
`const FPQueryExpressionRef inRef`

Concurrency Requirement:
This function is thread safe.

Description:
This function returns the end time (latest creation or deletion date) for which C-Clips are queried, as set by `FPQueryExpression_SetEndTime()`. End time is measured in milliseconds since 00:00:00 on January 1, 1970 (the UNIX/Java epoch). An end time of `-1` corresponds to the current time.

Parameters:
`const FPQueryExpressionRef inRef`
The reference to a query expression as returned from `FPQueryExpression_Create()`.

Example:
```
EndTime = FPQueryExpression_GetEndTime (myQueryExp);
```

Error Handling:
`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_GetStartTime

Syntax:

```
FPQueryExpression_GetStartTime (const
   FPQueryExpressionRef inRef)
```

Return Value: `FPLong`

Input Parameters: `const FPQueryExpressionRef inRef`

Concurrency Requirement: This function is thread safe.

Description: This function returns the start time (earliest creation or deletion date) for which C-Clips are queried, as set by `FPQueryExpression_SetStartTime()`. Start time is measured in milliseconds since 00:00:00 on January 1, 1970 (the UNIX/Java epoch).

Parameters: `const FPQueryExpressionRef inRef`
The reference to a query expression as returned from `FPQueryExpression_Create()`.

Example:

```
StartTime = FPQueryExpression_GetStartTime (myQueryExp);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_GetType

Syntax:
```
FPQueryExpression_GetType (const FPQueryExpressionRef
   inRef)
```

Return Value:
```
FPInt
```

Input Parameters:
```
const FPQueryExpressionRef inRef
```

Concurrency Requirement:
This function is thread safe.

Description:
This function returns the query type—existing C-Clips, deleted C-Clips (reflections), or both—as set by `FPQueryExpression_SetType()`. Possible values are:

- `FP_QUERY_TYPE_EXISTING` — Queries only existing C-Clips, not reflections.
- `FP_QUERY_TYPE_DELETED` — Queries only reflections.
- `FP_QUERY_TYPE_EXISTING | FP_QUERY_TYPE_DELETED` — Queries both existing C-Clips and reflections.

Parameters:
`const FPQueryExpressionRef inRef`
The reference to a query expression as returned from `FPQueryExpression_Create()`.

Example:
```
queryType = FPQueryExpression_GetType (myQueryExp);
```

Error Handling:
`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_IsFieldSelected

Syntax:

```
FPQueryExpression_IsFieldSelected (const
   FPQueryExpressionRef inRef, const char *inAttrName)
```

Return Value: `FPBool`

Input Parameters: `const FPQueryExpressionRef inRef, const char *inAttrName`

Concurrency Requirement: This function is thread safe.

Description: This function returns true if the description attribute is included in the query expression and false otherwise. Refer to *FPQueryExpression_SelectField* on page 1-231 for more information.

Parameters:

- `const FPQueryExpressionRef inRef`
  The reference to a query expression as returned from `FPQueryExpression_Create()`.
- `const char *inAttrName`
  The name of a C-Clip description attribute.

Example:

```
if (FPQueryExpression_IsFieldSelected (myQuery, Company))
   {...}
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_SelectField

Syntax:

```
FPQueryExpression_SelectField (const
   FPQueryExpressionRef inRef, const char *inAttrName)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryExpressionRef inRef, const char *inAttrName
```

Concurrency Requirement:

This function is thread safe.

Description:

This function includes a specified description attribute in the query result. No error is returned if the C-Clip does not contain the specified attribute.

Note: Including a description attribute does not affect the list of C-Clips returned by the query. Only the start time, end time, and query type affect which C-Clips are returned. Your application must process the query results to filter based on description attributes.

You can include both standard and user-defined attributes in a query expression. For information on user-defined attributes, refer to *FPClip_SetDescriptionAttribute* on page 1-116. Table 1-12, *Standard C-Clip Description Attributes*, lists the standard C-Clip attributes. Table 1-13, *Standard Reflection Description Attributes*, on page 1-232 lists the standard reflection attributes.

Table 1-12 Standard C-Clip Description Attributes

Attribute Name	Description
name	The name of the C-Clip. Refer to *FPClip_AuditedDelete* on page 1-48 and *FPClip_GetName* on page 1-96.
creation.date	The timestamp when the C-Clip was created. Refer to *FPClip_AuditedDelete* on page 1-48 and *FPClip_GetCreationDate* on page 1-89.
modification.date	The timestamp when the C-Clip was written to the cluster. Refer to *FPClip_Write* on page 1-80.
creation.profile	The profile used by the application that created the C-Clip.
modification.profile	The profile used by the application that wrote the modified C-Clip to the cluster.

Table 1-12 Standard C-Clip Description Attributes (continued)

Attribute Name	Description
numfiles	The number of referenced blobs. Refer to *FPClip_GetNumBlobs* on page 1-98.
totalsize	The total size of referenced blobs. This size does not include the CDF itself. Refer to *FPClip_GetRetentionClassName* on page 1-101.
prev.clip	The Clip ID of the C-Clip that was modified to create this C-Clipl
clip.naming.scheme	The C-Clip naming scheme: MD5 or MG.
numtags	The number of tags in the C-Clip. Refer to *FPClip_GetNumTags* on page 1-99.
sdk.version	The version of the SDK used to create the C-Clip. Refer to *FPPool_GetComponentVersion* on page 1-22.
retention.period	The retention period, in seconds. Only present if a retention period has been assigned to this C-Clip. Refer to *FPClip_SetRetentionPeriod* on page 1-72.
retention.class	The name of a retention class. Only present if a retention class has been assigned to this C-Clip. Refer to *FPClip_SetRetentionClass* on page 1-68.

Table 1-13 Standard Reflection Description Attributes

Attribute Name	Description
principal	The profile name used by the application that deleted the C-Clip.
incomingip	The IP address of the machine hosting the application that deleted the C-Clip.
creation.date	The timestamp of when the C-Clip was deleted (creation date of the reflection).

Table 1-13 Standard Reflection Description Attributes (continued)

Attribute Name	Description
deletedsize	The size of the deleted C-Clip and all referenced blobs. Note that this size may not represent the amount of data actually deleted, because blobs referenced by multiple C-Clips would not have been deleted.
reason	The audit string provided when the C-Clip was deleted. Refer to *FPClip_AuditedDelete* on page 1-48 for more information.
All standard and user-defined attributes of the deleted C-Clip.	A reflection retains all the standard and user-defined attributes of the original C-Clip.

Parameters:

- `const FPQueryExpressionRef inRef`
  The reference to a query expression as returned from `FPQueryExpression_Create()`.
- `const char *inAttrName`
  The name of a description attribute.

Example:

```
char* Company = "Company";
FPQueryExpression_SelectField(myQuery, Company);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_SetEndTime

Syntax:

```
FPQueryExpression_SetEndTime (const FPQueryExpressionRef
   inRef, const FPLong inTime)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryExpressionRef inRef, const FPLong inTime
```

Concurrency Requirement:

This function is thread safe.

Description:

This function sets the query condition that C-Clips retrieved by the query result have a creation or deletion time no later than the specified time. If an end time is not set, the default is -1 (current time). The storage time refers to the time that a C-Clip is actually stored on a cluster when the C-Clip was written or replicated. In the case of querying a replica cluster, the time used in a query for any replicated C-Clips is based on the storage time of when those C-Clips arrived at the replica cluster.

Parameters:

- `const FPQueryExpressionRef inRef`
  The reference to a query expression as returned from `FPQueryExpression_Create()`.
- `const FPLong inTime`
  The latest storage or deletion time (not inclusive) of C-Clips to be returned by the query. Specify the time in milliseconds since 00:00:00 on January 1, 1970 (the UNIX/Java epoch). The time is based on the UTC (Coordinated Universal Time, also known as GMT—Greenwich Mean Time) of the system clock of the Centera cluster that stores the C-Clips. A value of `-1` corresponds to the current time.

Note: For more information on timestamps, refer to Chapter 6, *Best Practices,* in the *Centera Programmer's Guide,* P/N 069001127.

Example:

```
FPQueryExpression_SetEndTime (myQueryExp, -1);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_SetStartTime

Syntax:

```
FPQueryExpression_SetStartTime (const
   FPQueryExpressionRef inRef, const FPLong inTime)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryExpressionRef inRef, const FPLong inTime
```

Concurrency Requirement:

This function is thread safe.

Description:

This function sets the query condition that C-Clips retrieved by the query have a storage or deletion time later than the specified time. If a start time is not set, the default is 0 (the earliest possible time). The storage time refers to the time that a C-Clip is actually stored on a cluster when the C-Clip was written or replicated. In the case of querying a replica cluster, the time used in a query for any replicated C-Clips is based on the storage time of when those C-Clips arrived at the replica cluster.

Parameters:

- `const FPQueryExpressionRef inRef`
  The reference to a query expression as returned from `FPQueryExpression_Create()`.
- `const FPLong inTime`
  The earliest storage or deletion time (inclusive) of C-Clips to be returned by the query. Specify the time in milliseconds since 00:00:00 on January 1, 1970 (the UNIX/Java epoch). The time is based on the UTC (Coordinated Universal Time, also known as GMT—Greenwich Mean Time) of the system clock of the Centera cluster that stores the C-Clips. If `inTime` is 0, the function queries all C-Clips stored or deleted on the cluster up to the specified stop time. Refer to `FPQueryExpression_SetEndTime()`.

Note: For more information on timestamps, refer to Chapter 6, *Best Practices,* in the *Centera Programmer's Guide,* P/N 069001127.

Example:

```
FPQueryExpression_SetStartTime (myQueryExp, 10000);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryExpression_SetType

Syntax:

```
FPQueryExpression_SetType (const FPQueryExpressionRef
   inRef, const FPInt inType)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryExpressionRef inRef, const FPInt inType
```

Concurrency Requirement:

This function is thread safe.

Description:

This function specifies what C-Clip types to query: existing C-Clips, deleted C-Clips (reflections), or both.

Parameters:

- `const FPQueryExpressionRef inRef`
  The reference to a query expression as returned from `FPQueryExpression_Create()`.
- `const FPInt inType`
  The type of C-Clips to query. Choices are:
  - `FP_QUERY_TYPE_EXISTING` — Query only existing C-Clips, not reflections.
  - `FP_QUERY_TYPE_DELETED` — Query only reflections.
  - `FP_QUERY_TYPE_EXISTING | FP_QUERY_TYPE_DELETED` — Query both existing C-Clips and reflections.

Example:

```
FPQueryExpression_SetType (myQueryExp,
   FP_QUERY_TYPE_EXISTING | FP_QUERY_TYPE_DELETED);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPPoolQuery_Close

Syntax: `FPPoolQuery_Close (const FPPoolQueryRef inPoolQueryRef)`

Return Value: `void`

Input Parameters: `const FPPoolQueryRef inPoolQueryRef`

Concurrency Requirement: This function is thread safe.

Description: This function closes a query on the pool and frees all associated (memory) resources. Note that calling this function on a pool query that has already been closed may produce unwanted results.

Parameters: `const FPPoolQueryRef inPoolQueryRef`
The reference to a pool query opened by `FPPoolQuery_Open()`.

Example:
```
FPPoolQuery_Close (myPoolQuery);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPPoolQuery_FetchResult

Syntax:

```
FPPoolQuery_FetchResult (const FPPoolQueryRef
   inPoolQueryRef, const FPInt inTimeout)
```

Return Value:

```
FPQueryResultRef
```

Input Parameters:

```
const FPPoolQueryRef inPoolQueryRef, const FPInt
   inTimeout
```

Concurrency Requirement:

This function is thread safe.

Description:

This function returns a query result. This function queries each C-Clip or reflection in time ascending order (from earliest to latest creation or deletion date). This function returns the query result in a QueryResult object. Process the query results with the `FPQueryResult_XXX()` functions.

Check the query status after each call to `FPPoolQuery_FetchResult()`. Refer to *FPQueryResult_GetResultCode* on page 1-246.

Call `FPQueryResult_Close()` after each call to `FPPoolQuery_FetchResult()` to free all associated (memory) resources.

Parameters:

- `const FPPoolQueryRef inPoolQueryRef`
  The reference to a pool query opened by `FPPoolQuery_Open()`.
- `const FPInt inTimeout`
  The time in milliseconds that the function waits for the next result. If `inTimeout` = -1, the function uses the default timeout of 120000 ms (2 minutes). The maximum timeout is 600000 (10 minutes).

  Note: Specifying a timeout value that is less than the default timeout of 120000 ms (2 minutes) may result in the system error code `FP_QUERY_RESULT_CODE_ERROR`.

Example:

```
myQueryResult = FPPoolQuery_FetchResult (myPoolQuery,
   600000);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)
- `FP_PARAM_ERR` (program logic error)

FPPoolQuery_GetPoolRef

Syntax:

```
FPPoolQuery_GetPoolRef (const FPPoolQueryRef
   inPoolQueryRef)
```

Return Value:

`FPPoolRef`

Input Parameters:

`const FPPoolQueryRef inPoolQueryRef`

Concurrency Requirement:

This function is thread safe.

Description:

This function returns the pool associated with a pool query. Refer to *FPPoolQuery_Close* on page 1-237 for more information.

Parameters:

`const FPPoolQueryRef inPoolQueryRef`
The reference to a pool query opened by `FPPoolQuery_Open()`.

Example:

```
myPool = FPPoolQuery_GetPoolRef (myPoolQuery);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_QUERYCLOSED_ERR` (client error)

FPPoolQuery_Open

Syntax:

```
FPPoolQuery_Open (const FPPoolRef inPoolRef,
   FPQueryExpressionRef inQueryExpressionRef)
```

Return Value:

`FPPoolQueryRef`

Input Parameters:

```
const FPPoolRef inPoolRef, FPQueryExpressionRef
   inQueryExpressionRef)
```

Concurrency Requirement:

This function is thread safe.

Description:

This function returns a reference to an open PoolQuery object. This function initiates a pool query. Iteratively call `FPPoolQuery_FetchResult()` to return query results.

The query conditions are specified by a query expression that you previously defined with `FPQueryExpression_xxx` functions. Refer to *FPQueryExpression_Close* on page 1-224 for more information.

Queries do not fail over by default in multicluster environments. Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on multicluster failover. To configure failover behavior, refer to *FPPool_SetGlobalOption* on page 1-38.

Note: The cluster allows the application to perform this call only if the "clip-enumeration" capability is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Note: The maximum number of parallel queries to a Centera cluster is 10.

When your query is complete, call `FPPoolQuery_Close()` to free any allocated resources.

Parameters:

- `const FPPoolRef inPoolRef`
  The reference to a pool opened by `FPPool_Open()`.
- `FPQueryExpressionRef inQueryExpressionRef`
  The reference to a query expression created by `FPQueryExpression_Create()`. The query expression defines the conditions for the query.

Example:

```
FPPoolQueryRef myPoolQuery = 0;

myPoolQuery = FPPoolQuery_Open(myPool, myQueryExp);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)
- `FP_NO_POOL_ERR` (network error)
- `FP_OUT_OF_MEMORY_ERR` (client error)
- Additional server errors

FPQueryResult_Close

Syntax:

```
FPQueryResult_Close (const FPQueryResultRef
   inQueryResultRef)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryResultRef inQueryResultRef
```

Concurrency Requirement:

This function is thread safe.

Description:

This function closes a query result as returned by `FPPoolQuery_FetchResult()` and frees all associated (memory) resources. Call this function after each call to `FPPoolQuery_FetchResult()`. Note that calling this function on a query result that has already been closed may produce unwanted results.

Parameters:

`const FPQueryResultRef inQueryResultRef`
A reference to a query result as returned by `FPPoolQuery_FetchResult()`.

Example:

```
FPQueryResult_Close (myQueryResult);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)(internal error)

FPQueryResult_GetClipID

Syntax:
```
FPQueryResult_GetClipID (const FPQueryResultRef
   inQueryResultRef, FPClipID outClipID)
```

Return Value:
```
void
```

Input Parameters:
```
const FPQueryResultRef inQueryResultRef
```

Output Parameters:
```
FPClipID outClipID
```

Concurrency Requirement: This function is thread safe.

Description: This function retrieves the ID of the C-Clip associated with the specified query result.

Parameters:

- `const FPQueryResultRef inQueryResultRef`
  A reference to a query result as returned by `FPPoolQuery_FetchResult()`.
- `FPClipID outClipID`
  The C-Clip ID associated with the query result.

Example:
```
FPClipID myClipID;
FPQueryResult_GetClipID (myQueryResult, myClipID);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PARAM_ERR` (program logic error)

FPQueryResult_GetField

Syntax:

```
FPQueryResult_GetField (const FPQueryResultRef
   inQueryResultRef, const char *inAttrName, char
   *outAttrValue, FPInt *ioAttrValueLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPQueryResultRef inQueryResultRef, const char
   *inAttrName, FPInt *ioAttrValueLen
```

Output Parameters:

```
char *outAttrValue, FPInt *ioAttrValueLen
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves the value of the given attribute from the C-Clip or reflection associated with the given query result. The application must have called `FPQueryExpression_SelectField()` to indicate that the attribute should be returned by the query.

Parameters:

- `const FPQueryResultRef inQueryResultRef`
  A reference to a query result as returned by `FPPoolQuery_FetchResult()`.
- `const char *inAttrName`
  The buffer that contains the name of the attribute for which the value is retrieved.
- `const char *outAttrValue`
  The buffer that will hold the attribute value.
- `FPInt *ioAttrValueLen`
  Input: The length in characters of `outAttrValue`.
  Output: The actual length of the attribute value, in characters, including the end-of-string character.

Example:

```
char vBuffer[1024];
FPInt len=sizeof(vBuffer);
FPQueryResult_GetField(myQueryResult, "Company",
   vBuffer, &len);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPQueryResult_GetResultCode

Syntax: `FPQueryResult_GetResultCode (const FPQueryResultRef inQueryResultRef)`

Return Value: `FPInt`

Input Parameters: `const FPQueryResultRef inQueryResultRef`

Concurrency Requirement: This function is thread safe.

Description: This function returns the result code of the specified query result, as returned by `FPPoolQuery_FetchResult()`. Possible values are:

- `FP_QUERY_RESULT_CODE_OK` — The query result is valid and can be processed by the application.
- `FP_QUERY_RESULT_CODE_INCOMPLETE` — The query result may be incomplete because one or more nodes could not be queried. It is recommended that the query be restarted from the last valid timestamp before `FP_QUERY_RESULT_CODE_INCOMPLETE` was returned.
- `FP_QUERY_RESULT_CODE_COMPLETE` — This value is always returned after `FP_QUERY_RESULT_CODE_INCOMPLETE`. Although the results of the query are complete, they may not be valid because one or more nodes could not be queried. This value indicates that all nodes can be queried again.
- `FP_QUERY_RESULT_CODE_END` — The query is finished; no more query results are expected.
- `FP_QUERY_RESULT_CODE_ABORT` — The query has aborted due to a problem on the cluster or because the start time for the query expression is later than the current server time. Check the start time and retry the query.
- `FP_QUERY_RESULT_CODE_PROGRESS` — The query is in progress.
- `FP_QUERY_RESULT_CODE_ERROR` — An error has been detected during the execution of this call. Check `FPPool_GetLastError()` to get the Centera error code. In the case where `FP_SOCKET_ERR` has been returned, call `FPPool_GetLastErrorInfo()` to check the OS-dependent error code. If this error refers to a timeout (for example, 10060 on Windows), it is recommended that you retry the query. Otherwise, return the error code when the call has been executed.

Note: A call to `FPPoolQuery_FetchResult` that specifies a timeout value less than the default timeout of 120000 ms (2 minutes) may result in this system error code.

Parameters: `const FPQueryResultRef inQueryResultRef`
A reference to a query result as returned by `FPPoolQuery_FetchResult()`.

Example:
```
ResultCode = FPQueryResult_GetResultCode (myQueryResult);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPQueryResult_GetTimestamp

Syntax:

```
FPQueryResult_GetTimestamp (const FPQueryResultRef
   inQueryResultRef)
```

Return Value: `FPLong`

Input Parameters: `const FPQueryResultRef inQueryResultRef`

Concurrency Requirement: This function is thread safe.

Description: This function returns the timestamp from the query result. For existing C-Clips, the timestamp is when the C-Clip was created on the cluster. For reflections (deleted C-Clips), the timestamp is when the C-Clip was deleted from the cluster. This function returns the timestamp in milliseconds since 00:00:00 on January 1, 1970 (the UNIX/Java epoch).

Parameters: `const FPQueryResultRef inQueryResultRef`
A reference to a query result as returned by `FPPoolQuery_FetchResult()`.

Example:

```
myTime = FPQueryResult_GetTimestamp (myQueryResult);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

FPQueryResult_GetType

Syntax:

```
FPQueryResult_GetType (const FPQueryResultRef
   inQueryResultRef)
```

Return Value:

```
FPInt
```

Input Parameters:

```
const FPQueryResultRef inQueryResultRef
```

Concurrency Requirement:

This function is thread safe.

Description:

This function returns the type of C-Clip, existing or deleted, associated with the query result. Possible values are:

- `FP_QUERY_TYPE_EXISTING` — The C-Clip exists on the cluster.
- `FP_QUERY_TYPE_DELETED` — The C-Clip has been deleted from the cluster.

Parameters:

`const FPQueryResultRef inQueryResultRef`
A reference to a query result as returned by `FPPoolQuery_FetchResult()`.

Example:

```
CClipType = FPQueryResult_GetType (myQueryResult);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)

Monitoring Functions

The Monitor API (MoPI) lets you gather monitoring information from the Centera server.

The MoPI is part of the FPLibrary shared library. The FPMonitor library authenticates itself to the server using the PAI module.

The information retrieved by the MoPI is in XML format and can be basic, statistical, or event-related. Refer to Appendix A, *Monitoring Information* for more details.

These are the MoPI functions:

- `FPEventCallback_Close`
- `FPEventCallback_RegisterForAllEvents`
- `FPMonitor_Close`
- `FPMonitor_GetAllStatistics`
- `FPMonitor_GetAllStatisticsStream`
- `FPMonitor_GetDiscovery`
- `FPMonitor_GetDiscoveryStream`
- `FPMonitor_Open`

When a monitoring transaction fails, the monitoring function fails over to another node with the access role, either one from the parameter list or one from the probe. If none of the nodes with the access role responds, the function returns an error.

FPEventCallback_Close

Syntax:

```
FPEventCallback_Close (const FPEventCallbackRef
   inRegister)
```

Return Value:

```
void
```

Input Parameters:

```
const FPEventCallbackRef inRegister
```

Concurrency Requirement:

This function is thread safe.

Description:

This function closes the gathering of events. The event connection to the server is stopped and all resources are deallocated.

Parameters:

```
const FPEventCallbackRef inRegister
```

The reference to an event callback as returned from the `FPEventCallback_RegisterForAllEvents` function.

Example:

```
FPEventCallback_Close(vRef);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OBJECTINUSE_ERR` (client error)

FPEventCallback_RegisterForAllEvents

Syntax:

```
FPEventCallback_RegisterForAllEvents (const FPMonitorRef
   inMonitor, FPStreamRef inStream)
```

Return Value:

`FPEventCallbackRef`

Input Parameters:

`const FPMonitorRef inMonitor, FPStreamRef inStream`

Concurrency Requirement:

This function is thread safe.

Description:

This function asynchronously registers the application to receive Centera events (alerts) in XML format. The registration remains active until the application closes the given monitor. As the stream callback functions will be called asynchronously, the application should not close the stream before closing the monitor.

The SDK sends—with an interval determined by the server—keep-alive monitoring packets to the cluster to ensure that the node with the access role is still online. The server answers with a keep-alive reply.

If the node with the access role is offline, the alert-receiving thread fails over to another node with the access role. If all nodes with the access role are offline, a special SDK-alert is pushed to the output stream.

This function returns a reference to an event callback. This callback has to be used to close the callback registration using the `FPEventCallback_Close` function.

Refer to Appendix A, *Monitoring Information*, for the syntax and a sample of the event (alert) information.

Note: The server allows the application to perform this call if the server capability "monitor" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

Parameters:

- `const FPMonitorRef inMonitor`
  The reference to a monitor object as returned from `FPMonitor_Open()`. The reference may also be `NULL`.
- `FPStreamRef inStream`
  The reference to a stream (as returned from the functions `FPStream_CreateXXX()` or `FPStream_CreateGenericStream()`).

Example:

```
FPStreamRef vStream=CreateMyStream();
FPEventCallbackRef
   vRef=FPEventCallback_RegisterForAllEvents(vMonitor,
   vStream);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPMonitor_Close

Syntax: `FPMonitor_Close (const FPMonitorRef inMonitor)`

Return Value: `void`

Input Parameters: `const FPMonitorRef inMonitor`

Concurrency Requirement: This function is thread safe.

Description: This function closes the given monitor object and frees all related (memory) resources. Note that calling this function on a monitor object that has already been closed may produce unwanted results.

Parameters: `const FPMonitorRef inMonitor`
The reference to a monitor object as returned from `FPMonitor_Open()`.

Example:
```
FPMonitor_Close(vMonitor);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_OBJECTINUSE_ERR` (client error)

FPMonitor_GetAllStatistics

Syntax:

```
FPMonitor_GetAllStatistics (const FPMonitorRef inMonitor,
   char *outData, FPInt *ioDataLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPMonitorRef inMonitor, FPInt *ioDataLen
```

Output Parameters:

```
char *outData, FPInt *ioDataLen
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves all available statistical information about the Centera cluster in XML format and writes it to the given buffer. The monitor object has been opened with `FPMonitor_Open()`.

Server information that constantly changes is referred to as statistical information. Refer to Appendix A, *Monitoring Information*, for the syntax and a sample of the statistical information.

Note: The server allows the application to perform this call if the server capability "monitor" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

If the application retries the call, for example because the buffer was not large enough, it is not guaranteed that the new data will be identical to the data as retrieved by the first call.

Parameters:

- `const FPMonitorRef inMonitor`
  The reference to a monitor object as returned from `FPMonitor_Open()`. The reference may also be `NULL`.
- `char *outData`
  `outData` is the memory buffer that will store the statistical information.
- `FPInt *ioDataLen`
  Input: The reserved length, in characters, of the `outData` buffer.
  Output: The actual length of the statistical information, in characters, including the end-of-string character.

Example:

```
char vBuffer[10*1024];
FPInt l=sizeof(vBuffer);
FPMonitor_GetAllStatistics(vMonitor, vBuffer, &l);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPMonitor_GetAllStatisticsStream

Syntax:

```
FPMonitor_GetAllStatisticsStream (const FPMonitorRef
   inMonitor, FPStreamRef inStream)
```

Return Value:

```
void
```

Input Parameters:

```
const FPMonitorRef inMonitor, FPStreamRef inStream
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves all available statistical information about the Centera cluster in XML format and writes it to the given stream. The monitor object has been opened with `FPMonitor_Open()`.

Note: The server allows the application to perform this call if the server capability "monitor" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

If the application retries the call, for example because the buffer was not large enough, it is not guaranteed that the new data will be identical to the data as retrieved by the first call.

Parameters:

- `const FPMonitorRef inMonitor`
  The reference to a monitor object as returned from `FPMonitor_Open()`. The reference may also be `NULL`.
- `FPStreamRef inStream`
  The reference to a stream (as returned from the functions `FPStream_CreateXXX()` or `FPStream_CreateGenericStream()`).

Example:

```
FPStreamRef vStream=FPStream_CreateToStdio();
FPMonitor_GetAllStatisticsStream(vMonitor, vStream);
FPStream_Close(vStream);
```

Error Handling:

`FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)

- FP_NO_SOCKET_AVAIL_ERR (network error)
- FP_PROTOCOL_ERR (internal error)
- FP_UNKNOWN_OPTION (internal error)
- FP_SERVER_ERR (server error)
- FP_CONTROLFIELD_ERR (server error)
- FP_NOT_RECEIVE_REPLY_ERR (network error)
- FP_SEGDATA_ERR (internal error)
- FP_BLOBBUSY_ERR (server error)
- FP_SERVER_NOTREADY_ERR (server error)
- FP_PROBEPACKET_ERR (internal error)
- FP_CLIPCLOSED_ERR (program logic error)
- FP_POOLCLOSED_ERR (program logic error)
- FP_OPERATION_NOT_ALLOWED (client error)
- FP_WRONG_STREAM_ERR (client error)

FPMonitor_GetDiscovery

Syntax:

```
FPMonitor_GetDiscovery (const FPMonitorRef inMonitor,
   char *outData, FPInt *ioDataLen)
```

Return Value:

```
void
```

Input Parameters:

```
const FPMonitorRef inMonitor, FPInt *ioDataLen
```

Output Parameters:

```
char *outData, FPInt *ioDataLen
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves discovery information about the Centera cluster in XML format and writes it to the given buffer. The monitor object has been opened with `FPMonitor_Open()`.

General server information such as number of nodes and capacity is referred to as discovery information. Refer to Appendix A, *Monitoring Information*, for the syntax and a sample of the discovery information.

Note: The server allows the application to perform this call if the server capability "monitor" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

If the application retries the call, for example because the buffer was not large enough, it is not guaranteed that the new data will be identical to the data as retrieved by the first call.

Parameters:

- `const FPMonitorRef inMonitor`
  The reference to a monitor object as returned from `FPMonitor_Open()`. The reference may also be `NULL`.
- `char *outData`
  `outData` is the memory buffer that will store the discovery information.
- `FPInt *ioDataLen`
  Input: The reserved length, in characters, of the `outData` buffer.
  Output: The actual length of the discovery information, in characters, including the end-of-string character.

Example:

```
char vBuffer[10*1024];
FPInt l=sizeof(vBuffer);
FPMonitor_GetDiscovery(vMonitor, vBuffer, &l);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPMonitor_GetDiscoveryStream

Syntax:

```
FPMonitor_GetDiscoveryStream (const FPMonitorRef
   inMonitor, FPStreamRef inStream)
```

Return Value:

```
void
```

Input Parameters:

```
const FPMonitorRef inMonitor, FPStreamRef inStream
```

Concurrency Requirement:

This function is thread safe.

Description:

This function retrieves discovery information about the Centera cluster in XML format and writes it to the given stream. The monitor object has been opened with `FPMonitor_Open()`.

Note: The server allows the application to perform this call if the server capability "monitor" is enabled. If this capability is disabled, the error `FP_OPERATION_NOT_ALLOWED` is returned.

If the application retries the call, it is not guaranteed that the new data will be identical to the data as retrieved by the first call.

Parameters:

- `const FPMonitorRef inMonitor`
  The reference to a monitor object as returned from `FPMonitor_Open()`. The reference may also be `NULL`.
- `FPStreamRef inStream`
  The reference to a stream (as returned from the functions `FPStream_CreateXXX()` or `FPStream_CreateGenericStream()`).

Example:

```
FPStreamRef vStream=FPStream_CreateToStdio();
FPMonitor_GetDiscoveryStream(vMonitor, vStream);
FPStream_Close(vStream);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error or internal error; verify your code before contacting the EMC Customer Support Center)
- `FP_ATTR_NOT_FOUND_ERR` (internal error)
- `FP_TAG_HAS_NO_DATA_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_FILE_NOT_STORED_ERR` (program logic error)
- `FP_NO_POOL_ERR` (network error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_UNKNOWN_OPTION` (internal error)
- `FP_SERVER_ERR` (server error)
- `FP_CONTROLFIELD_ERR` (server error)
- `FP_NOT_RECEIVE_REPLY_ERR` (network error)
- `FP_SEGDATA_ERR` (internal error)
- `FP_BLOBBUSY_ERR` (server error)
- `FP_SERVER_NOTREADY_ERR` (server error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_CLIPCLOSED_ERR` (program logic error)
- `FP_POOLCLOSED_ERR` (program logic error)
- `FP_OPERATION_NOT_ALLOWED` (client error)
- `FP_WRONG_STREAM_ERR` (client error)

FPMonitor_Open

Syntax: `FPMonitor_Open (const char *inClusterAddress)`

Return Value: `FPMonitorRef`

Input Parameters: `const char *inClusterAddress`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function creates a new object to monitor the state of the Centera server. The monitoring functions operate on the first available IP address in the given list of cluster addresses.

This function checks the availability of a node with the access role using the UDP Probe transaction. The reply of this transaction contains the clusterID that is needed for the authentication phase. This function uses the PAI module to retrieve the authentication information.

This function returns a reference to an FPMonitor object. If no connection could be made, the function returns `NULL`.

Parameters: `const char *inClusterAddress`
A list of comma-separated IP addresses of nodes with the access role belonging to one cluster. Information for the PAI module should be added using a question mark as delimiter; refer to the example below.

Example:

```
myClusterAddress = "10.62.69.153?c:\\centera\rwe.pea";
myMonitor = FPMonitor_Open (myClusterAddress);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_WRONG_REFERENCE_ERR` (program logic error)
- `FP_PROTOCOL_ERR` (internal error)
- `FP_NO_SOCKET_AVAIL_ERR` (network error)
- `FP_PROBEPACKET_ERR` (internal error)
- `FP_NO_POOL_ERR` (network error
- `FP_ACCESSNODE_ERR` (network error)
- `FP_AUTHENTICATION_FAILED_ERR` (server error)

Time Functions

This section describes the following time formats used by the SDK to convert Centera time strings to integral time values marking the time since the epoch 1 January 1970 00:00:00.000 GMT. These API calls represent integral units in either seconds or milliseconds.

The SDK supports two string formats when converting the integral values to time strings. These string formats are defined as options `FP_OPTION_MILLISECONDS_STRING` and `FP_OPTION_SECONDS_STRING` with a flag argument. The `inOptions` argument produces a time string with or without a milliseconds field.

- `FPTime_MillisecondsToString`
- `FPTime_SecondsToString`
- `FPTime_StringToMilliseconds`
- `FPTime_StringToSeconds`

FPTime_MillisecondsToString

Syntax:

```
FPTime_MillisecondsToString (FPLong inTime,
   char* outString, int* ioStringLen, int inOptions)
```

Return Value: `void`

Input Parameters: `FPLong inTime`

Output Parameters: `char* outString, int* ioStringLen, int inOptions`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function converts an `FPLong` that represents the number of milliseconds since the epoch 1 January 1970 00:00:00.000 GMT (Greenwich Mean Time) to a date-time string of the form `YYYY.MM.DD hh:mm:ss.ms GMT`. The function does not support time strings before the epoch.

For example, March 31, 2005 might be expressed as `2005.03.31 15:14:30.585 GMT`.

Parameters:

- `const FPLong inTime`
  The reference to the number of milliseconds since the epoch.
- `char* outString`
  A pointer to a user-allocated buffer that holds the resulting time string.
- `int* ioStringLen`
  Input: The pointer to an integer that holds the length of the buffer allocated for `outString`.
  Output: The integer is updated to hold the length of the resulting string.
- `int inOptions`
  Specify either `FP_OPTION_MILLISECONDS_STRING` to produce a string containing a milliseconds field or `FP_OPTION_SECONDS_STRING` to produce a string without a milliseconds field.

Example:

```
char vTimeString[MAX_STRING_LEN];
FPInt vTimeStringLen = MAX_STRING_LEN;
FPTime_LongToString(vTimeInSeconds, vTimeString,
   &vTimeStringLen);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)

FPTime_SecondsToString

Syntax:

```
FPTime_SecondsToString (FPLong inTime,
   char* outString, int* ioStringLen, int inOptions)
```

Return Value:

```
void
```

Input Parameters:

```
FPLong inTime, int inOptions
```

Output Parameters:

```
char* outString, int* ioStringLen
```

Concurrency Requirement:

This function is thread safe.

Unicode Support:

This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description:

This function converts an `FPLong` that represents the number of seconds since the epoch 1 January 1970 00:00:00.000 GMT to a date-time string of the form `YYYY.MM.DD hh:mm:ss.ms GMT`. The function does not support time strings before the epoch.

Parameters:

- `const FPLong inTime`
  The reference to the number of seconds since the epoch.
- `char* outString`
  A pointer to a user-allocated buffer that holds the resulting time string.
- `int* ioStringLen`
  Input: The pointer to an integer that holds the length of the buffer allocated for `outString`.
  Output: The integer is updated to hold the length of the resulting string.
- `int inOptions`
  Specify either `FP_OPTION_MILLISECONDS_STRING` to produce a string containing a milliseconds field or `FP_OPTION_SECONDS_STRING` to produce a string without a milliseconds field.

Example:

```
char vTimeString[MAX_STRING_LEN];
FPInt vTimeStringLen = MAX_STRING_LEN;
FPPool_GetClusterTime(vPool, vTimeString,
   &vTimeStringLen);
FPLong vTimeInSeconds = FPTime_StringToLong(vTimeString);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)

FPTime_StringToMilliseconds

Syntax: `FPTime_StringToMilliseconds (const char* inTime)`

Return Value: `FPLong`

Input Parameters: `const char* inTime`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function converts a date/time string to an `FPLong`, a return value that represents the number of milliseconds since the epoch 1 January 1970 00:00:00.000 GMT. The function does not support time strings before the epoch.

Parameters:

- `inTime`
  The time string of the format in `YYYY.MM.DD HH:MM:SS[.ms] GMT`, in which the milliseconds field is optional.

  Note: The milliseconds field is optional.

Example:

```
char vTimeString[MAX_STRING_LEN];
FPInt vTimeStringLen = MAX_STRING_LEN;
FPPool_GetClusterTime(vPool, vTimeString,
   &vTimeStringLen);
FPLong vTimeInSeconds =
   FPTime_StringToMilliseconds(vTimeString);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)

FPTime_StringToSeconds

Syntax: `FPTime_StringToSeconds (const char* inTime)`

Return Value: `FPLong`

Input Parameters: `const char *inTime`

Concurrency Requirement: This function is thread safe.

Unicode Support: This function has variants that support wide character and 8, 16, and 32-bit Unicode. For more information, see *Unicode and Wide Character Support* on page 1-4.

Description: This function converts a date/time string to an `FPLong`, a return value that represents the number of seconds since the epoch 1 January 1970 00:00:00.000 GMT. The function does not support time strings before the epoch.

Parameters:

- `inTime`
  The time string of the format in `YYYY.MM.DD HH:MM:SS[.ms] GMT`, in which the milliseconds field is optional.

  Note: The milliseconds field is optional.

Example:

```
char vTimeString[MAX_STRING_LEN];
FPInt vTimeStringLen = MAX_STRING_LEN;
FPPool_GetClusterTime(vPool, vTimeString,
   &vTimeStringLen);
FPLong vTimeInSeconds =
   FPTime_StringToSeconds(vTimeString);
```

Error Handling: `FPPool_GetLastError()` returns `ENOERR` if successful. If unsuccessful, the following is a partial list of possible errors:

- `FP_PARAM_ERR` (program logic error)
- `FP_OPERATION_NOT_SUPPORTED` (program logic error)

Error Codes

The API reports both Centera-specific and operating-system errors. If the API returns an error code that is not listed in Table 1-14, *Error Codes*, refer to a list of platform-specific error codes (Windows error code 10055, for example, means that no buffer space is available).

Note that all Centera error codes are negative values.

Refer to the *Centera Programmer's Guide*, P/N 069001127, for more information on error handling. Also refer to *FPPool_GetLastError* on page 1-26 and *FPPool_GetLastErrorInfo* on page 1-27.

If you want to access the errors described in this section, you must include `FPErrors.h`.

Table 1-14 Error Codes

Value	Error Name	Description and Action
-10001	FP_INVALID_NAME	The name that you have used is not XML compliant.
-10002	FP_UNKNOWN_OPTION	You have used an unknown option name with `FPPool_SetIntOption()` or `FPPool_GetIntOption()`.
-10003	FP_NOT_SEND_REQUEST_ERR	An error occurred when you sent a request to the server. This internal error was generated because the server could not accept the request packet. Verify all LAN connections and try again.
-10004	FP_NOT_RECEIVE_REPLY_ERR	No reply was received from the server. This internal error was generated because the server did not send a reply to the request packet. Verify all LAN connections and try again.
-10005	FP_SERVER_ERR	The server reports an error. An internal error on the server occurred. Try again.
-10006	FP_PARAM_ERR	You have used an incorrect or unknown parameter. **Example:** Is a string-variable too long, `null`, or empty when it should not be? Does a parameter have a limited set of values? Check each parameter in your code.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10007	FP_PATH_NOT_FOUND_ERR	This path does not correspond to a file or directory on the client system. The path in one of your parameters does not point to an existing file or directory. Verify the path in your code.
-10008	FP_CONTROLFIELD_ERR	The server reports that the operation generated a "Controlfield missing" error. This internal error was generated because the required control field was not found. Try again. (Obsolete from v2.0.)
-10009	FP_SEGDATA_ERR	The server reports that the operation generated a "Segdatafield missing" error. This internal error was generated because the required field containing the blob data was not found in the packet. Try again. (Obsolete from v2.0.)
-10010	FP_DUPLICATE_FILE_ERR	A duplicate CA already exists on the server. If you did not enable duplicate file detection, verify that you have not already stored this data and try again.
-10011	FP_OFFSET_FIELD_ERR	The server reports that the operation generated an "Offsetfield missing" error. This internal error was generated because the offset field was not found in the packet. Try again. (Obsolete from v2.0.)
-10012	FP_OPERATION_NOT_SUPPORTED	This operation is not supported. If `FPClip_Write()`, `FPTag_GetSibling()`, `FPTag_GetPrevSibling()`, `FPTag_GetFirstChild()` or `FPTag_Delete()` returned this error, then this operation is not supported for C-Clips opened in 'flat' mode. If `FPStream` returned this error, then you are trying to perform an operation that is not supported by that stream.
-10013	FP_ACK_NOT_RCV_ERR	A write acknowledgement was not received. Verify your LAN connections and try again.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10014	FP_FILE_NOT_STORED_ERR	Could not write the blob to the server OR could not find the blob on the server. This internal error was generated because the store operation of the blob was not successful. Verify that the original data was correctly stored, verify your LAN connections and try again.
-10015	FP_NUMLOC_FIELD_ERR	The server reports that the operation generated a "Numlockfield missing" error. This internal error was generated because the numlock field was not found in the packet. Try again. (Obsolete from v2.0.)
-10016	FP_SECTION_NOT_FOUND_ERR	The GetSection request could not retrieve the defined section tag. This internal error was generated because a required section is missing in the CDF. Verify the content of your code and try again. (Obsolete from v2.0.)
-10017	FP_TAG_NOT_FOUND_ERR	The referenced tag could not be found in the CDF. This internal error was generated because information is missing from the description section in the CDF. Verify the content of your code and try again.
-10018	FP_ATTR_NOT_FOUND_ERR	Could not find an attribute with that name. If `FPTag_GetXXXAttribute()` returned this error, then the attribute was not found in the tag. If `FPTag_GetIndexAttribute()` returned this error, then the index parameter is larger than the number of attributes in the tag.
-10019	FP_WRONG_REFERENCE_ERR	The reference that you have used is invalid. The reference was not opened, already closed, or not of the correct type.
-10020	FP_NO_POOL_ERR	It was not possible to establish a connection with a cluster. The server could not be located. This means that none of the IP addresses could be used to open a connection to the server or that no cluster could be found that has the required capability. Verify your LAN connections, server settings, and try again.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10021	FP_CLIP_NOT_FOUND_ERR	Could not find the referenced C-Clip in the cluster. Returned by FPClip_Open(), it means the CDF could not be found on the server. Verify that the original data was correctly stored and try again.
-10022	FP_TAGTREE_ERR	An error exists in the tag tree. Verify the content of your code and try again.
-10023	FP_ISNOT_DIRECTORY_ERR	A path to a file has been given but a path to a directory is expected. Verify the path to the data and try again.
-10024	FP_UNEXPECTEDTAG_ERR	Either a “file“ or “folder“ tag was expected but not given. An unexpected tag was found when retrieving the CDF. The CDF is probably corrupt.
-10025	FP_TAG_READONLY_ERR	The tag cannot be changed or deleted (it is probably a top tag). Verify your program logic.
-10026	FP_OUT_OF_BOUNDS_ERR	The options parameter is out of bounds. One of the function parameters exceeds its preset limits. Verify each parameter in your code.
-10027	FP_FILESYS_ERR	A file system error occurred, for example an incorrect path was given, or you are trying to open an unknown file or a file in the wrong mode. Verify the path and try again.
-10029	FP_STACK_DEPTH_ERR	You have exceeded the nested tag limit. Review the structure of your content description and try again. Deprecated.
-10030	FP_TAG_HAS_NO_DATA_ERR	You are trying to access blob data of a tag that does not contain blob data.
-10031	FP_VERSION_ERR	The C-Clip has been created using a more recent version of the client software than you are using. Upgrade to the latest version.
-10032	FP_MULTI_BLOB_ERR	The tag already has data associated with it. You need to create a new tag to store the new data or delete this tag and recreate it and try again.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10033	FP_PROTOCOL_ERR	You have used an unknown protocol option (Only HPP is supported). Verify the parameters in your code. It is also possible that an internal communication error occurred between the server and client. If you have verified your code and the problem persists then you need to upgrade to the latest client and server versions.
-10034	FP_NO_SOCKET_AVAIL_ERR	No new network socket is available for the transaction. Reduce the number of open transactions between the client and the server or use the function FPPool_SetGlobalOption() to increase the number of available sockets with FP_OPTION_MAXCONNECTIONS.
-10035	FP_BLOBIDFIELD_ERR	A BlobID field (the Content Address) was expected but not given. Upgrade to the latest client and server versions. (Obsolete from v2.0.)
-10036	FP_BLOBIDMISMATCH_ERR	The blob is corrupt: a BlobID mismatch occurred between the client and server. The Content Address calculation on the client and the server has returned different results. The blob is corrupt. If FPClip_Open() returns this error, it means the blob data or metadata of the C-Clip is corrupt and cannot be decoded.
-10037	FP_PROBEPACKET_ERR	The probe packet does not contain valid server addresses. Upgrade to the latest client and server versions. (Obsolete from v2.0.)
-10038	FP_CLIPCLOSED_ERR	(Java only.) You tried to perform an operation on a closed C-Clip. This operation requires access to an open C-Clip. Verify your code and try again.
-10039	FP_POOLCLOSED_ERR	(Java only.) You tried to perform an operation on a closed pool. This operation requires access to an open pool. Verify your code and LAN connections and try again.
-10040	FP_BLOBBUSY_ERR	The blob on the cluster is busy and cannot be read from or written to. You tried to read from or write to a blob that is currently busy with another read/write operation. Try again.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10041	FP_SERVER_NOTREADY_ERR	The server is not ready yet. This error can occur when a client tries to connect to the server to execute an operation and the nodes with the access role are running but the nodes with the storage role have not been initialized yet. This error can also occur when not enough mirror groups are found on the server. Try again.
-10042	FP_SERVER_NO_CAPACITY_ERR	The server has no capacity to store data. Enlarge the server's capacity and try again.
-10043	FP_DUPLICATE_ID_ERR	The application passed in a sequence ID that was previously used.
-10101	FP_SOCKET_ERR	An error on the network socket occurred. Verify the network.
-10102	FP_PACKETDATA_ERR	The data packet contains wrong data. Verify the network, the version of the server or try again later.
-10103	FP_ACCESSNODE_ERR	No node with the access role can be found. Verify the IP addresses provided with `FPPool_Open()`.
-10151	FP_OPCODE_FIELD_ERR	The Query Opcode field is missing from the packet.
-10152	FP_PACKET_FIELD_MISSING_ERR	The packet field is missing.
-10153	FP_AUTHENTICATION_FAILED_ERR	Authentication to get access to the server failed. Check the profile name and secret.
-10154	FP_UNKNOWN_AUTH_SCHEME_ERR	An unknown authentication scheme has been used.
-10155	FP_UNKNOWN_AUTH_PROTOCOL_ERR	An unknown authentication protocol has been used.
-10156	FP_TRANSACTION_FAILED_ERR	Transaction on the server failed. `FPClip_Delete()` or `FPClip_AuditedDelete()` could not delete the complete C-Clip because of server problems. Try again later.
-10157	FP_PROFILECLIPID_NOTFOUND_ERR	No profile clip was found.
-10158	FP_ADVANCED_RETENTION_DISABLED_ERR	The Advanced Retention Management feature is not licensed or enabled for event-based retention (EBR) and retention hold.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10159	FP_NON_EBR_CLIP_ERR	An attempt was made to trigger an EBR event on a C-Clip that is not eligible to receive an event.
-10160	FP_EBR_OVERRIDE_ERR	An attempt was made to trigger or enable the event-based retention period/class of a C-Clip a second time. You can set EBR information only once.
-10161	FP_NO_EBR_EVENT_ERR	The C-Clip is under event-based retention protection and cannot be deleted.
-10162	FP_RETENTION_OUT_OF_BOUNDS_ERR	The event-based retention period being set does not meet the minimum/maximum rule.
-10163	FP_RETENTION_HOLD_COUNT_ERR	The number of retention holds exceeds the limit of 100.
-10201	FP_OPERATION_REQUIRES_MARK	The application requires marker support but the stream does not provide that.
-10202	FP_QUERYCLOSED_ERR	The FPQuery for this object is already closed. (Java only).
-10203	FP_WRONG_STREAM_ERR	The function expects an input stream and gets an output stream or vice-versa.
-10204	FP_OPERATION_NOT_ALLOWED	The use of this operation is restricted or this operation is not allowed because the server capability is false.
-10205	FP_SDK_INTERNAL_ERR	An SDK internal programming error has been detected.
-10206	FP_OUT_OF_MEMORY_ERR	The system ran out of memory. Check the system's capacity.
-10207	FP_OBJECTINUSE_ERR	Cannot close the object because it is in use. Check your code.
-10208	FP_NOTYET_OPEN_ERR	The object is not yet opened. Check your code.
-10209	FP_STREAM_ERR	An error occurred in the generic stream. Check your code.
-10210	FP_TAG_CLOSED_ERR	The FPTag for this object is already closed. (Java only.)
-10211	FP_THREAD_ERR	An error occurred while creating a background thread.
-10212	FP_PROBE_TIME_EXPIRED_ERR	The probe limit time was reached.
-10213	FP_PROFILECLIPID_WRITE_ERR	There was an error while storing the profile clip ID.
-10214	FP_INVALID_XML_ERR	The specified string is not valid XML.

Table 1-14 Error Codes (continued)

Value	Error Name	Description and Action
-10215	FP_UNABLE_TO_GET_LAST_ERROR	The call to FPPool_GetLastError() or FPPool_GetLastErrorInfo() failed. The error status of the previous function call is unknown; the previous call may have succeeded.

Logging

To support application debugging, the SDK provides thread-safe logging of its activities without creating any new threads. The SDK logging system supports all platforms and generates log files based on either an XML or tab-delimited format.

The log file contains the following information:

- **Timestamp**—The current time of the client system when the message was logged. This timestamp records the number of milliseconds since the epoch 1 January 1970 00:00:00.000.
- **Message type**—The verbosity level of the message to indicate what type of information is to be included in the log (for example, `log` or `debug`).
- **Thread ID**—The ID of the thread that logged the message.
- **Component field**—A two-part field that first displays the API component that logged the message, followed by the root SDK API call in the stack that generated the log message.

Environment Variables

To enable logging and define specific logging features, you set the following environment variables.

FP_LOGPATH Setting `FP_LOGPATH` enables logging. The value for `FP_LOGPATH` is the path and file name containing the log.

FP_LOGKEEP Setting `FP_LOGKEEP` defines how the log file is to be generated. The value can be either `OVERWRITE`, `APPEND`, or `CREATE`. If a log file does not exist, the SDK automatically creates a log file as named in `FP_LOGPATH` regardless of the set value. If the log file exists, the SDK API uses the set value to perform one of the following actions:

- `OVERWRITE`: (Default) Replaces the existing log file by writing over it.
- `APPEND`: Adds the logging information to the end of the existing log in the log file.
- `CREATE`: Creates a new log file name by appending a timestamp to the end of the existing log file name and writes log messages to the new file.

FP_LOGLEVEL

Setting FP_LOGLEVEL assigns the log's verbosity of messages or which types of messages are to be included in the log, as follows.

1 - Error messages only

2 - Warning and Error messages

3 - Log, Warning, and Error messages

4 - Debug, Log, Warning, and Error messages

Note: If you do not use the FP_LOGLEVEL environment variable, the default message level is set to 3.

FP_LOGFILTER

Setting FP_LOGFILTER determines which components to represent in the written log. If FP_LOGFILTER is set, you designate the components in a comma-separated list, otherwise, the default is to log all components. The possible components you can list are POOL, RETRY, XML, API, NET, TRANSACTION, PACKET, EXCEPTION, REFS, MOPI, STREAM, CSOD, CSO, MD5, SCRATCH, and ALL.

FP_LOGFORMAT

Setting FP_LOGFORMAT determines the format of the generated log file—either XML or TAB. By default, the SDK API generates a tab-delimited log file, including upon error.

XML log format per line

```
<log level tID=thread id sec=timestamp comp=component
   API=high level API><![CDATA[message]]></log level>
```

If FP_LOGFORMAT is set to TAB, the SDK API generates a tab-delimited log file.

Tab-delimited log format per line

```
timestamp time string message type thread id component
   message
```

Example of an XML-formatted log

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
   <SDKLog SDKVersion="3.1.000"
   branch="falloj_U05_G00_INT_V3.1.000">

<log tID="1220" sec="1117217491436" comp="API" API="Start
   FPPool_Open8(%s)"><![CDATA[Start
   FPPool_Open8(10.241.60.26)]]></log>

<debug tID="1220" sec="1117217491451" comp="RETRY"
   API="FPPool_Open8(%s)"><![CDATA[
cluster becoming unavailable
]]></debug>
.
.
.
<log tID="1220" sec="1117217491592" comp="API" API="End
   FPPool_Open8(%s)"><![CDATA[
End FPPool_Open8(10.241.60.26)
]]></log>
</SDKLog>
```

A

Monitoring Information

This appendix lists the syntax and provides samples of XML files that are retrieved by the MoPI functions.

Discovery Information

This section lists the syntax and a sample of the XML file that can be retrieved by `FPMonitor_GetDiscovery`.

Syntax Discovery

```
<!ELEMENT applicationcontext ( client, securityprofile ) >

<!ELEMENT applicationcontexts ( applicationcontext* ) >

<!ELEMENT ats EMPTY >
<!ATTLIST ats powersource CDATA #REQUIRED >
<!ATTLIST ats status CDATA #REQUIRED >

<!ELEMENT cabinet ( ats?, cubeswitches, nodes ) >
<!ATTLIST cabinet id CDATA #REQUIRED >
<!ATTLIST cabinet availablerawcapacity CDATA #REQUIRED >
<!ATTLIST cabinet offlinerawcapacity CDATA #REQUIRED >
<!ATTLIST cabinet totalrawcapacity CDATA #REQUIRED >
<!ATTLIST cabinet usedrawcapacity CDATA #REQUIRED >

<!ELEMENT cabinets ( cabinet* ) >

<!ELEMENT capabilities ( capability* ) >

<!ELEMENT capability EMPTY >
<!ATTLIST capability enabled CDATA #REQUIRED >
<!ATTLIST capability name CDATA #REQUIRED >

<!ELEMENT client EMPTY >
<!ATTLIST client ip CDATA #IMPLIED >

<!ELEMENT cluster ( cabinets, rootswitches, services, pools, licenses
   ) >
<!ATTLIST cluster availablerawcapacity CDATA #REQUIRED >
<!ATTLIST cluster clusterid CDATA #REQUIRED >
<!ATTLIST cluster compliancemode CDATA #REQUIRED >
<!ATTLIST cluster contactemail CDATA #REQUIRED >
<!ATTLIST cluster contactname CDATA #REQUIRED >
<!ATTLIST cluster groomingvisit CDATA #REQUIRED >
<!ATTLIST cluster location CDATA #REQUIRED >
<!ATTLIST cluster name CDATA #REQUIRED >
<!ATTLIST cluster offlinerawcapacity CDATA #REQUIRED >
<!ATTLIST cluster protectionschemes CDATA #REQUIRED >
<!ATTLIST cluster serial CDATA #REQUIRED >
<!ATTLIST cluster serviceid CDATA #REQUIRED >
<!ATTLIST cluster serviceinfo CDATA #REQUIRED >
<!ATTLIST cluster siteid CDATA #REQUIRED >
```

```
<!ATTLIST cluster softwareversion CDATA #REQUIRED >
<!ATTLIST cluster totalrawcapacity CDATA #REQUIRED >
<!ATTLIST cluster usedrawcapacity CDATA #REQUIRED >

<!ELEMENT cubeswitch EMPTY >
<!ATTLIST cubeswitch description CDATA #REQUIRED >
<!ATTLIST cubeswitch ip CDATA #REQUIRED >
<!ATTLIST cubeswitch mac CDATA #REQUIRED >
<!ATTLIST cubeswitch name CDATA #REQUIRED >
<!ATTLIST cubeswitch rail CDATA #REQUIRED >
<!ATTLIST cubeswitch serial CDATA #REQUIRED >
<!ATTLIST cubeswitch status CDATA #REQUIRED >

<!ELEMENT cubeswitches ( cubeswitch* ) >

<!ELEMENT discovery ( format?, cluster? ) >

<!ELEMENT format EMPTY >
<!ATTLIST format version CDATA #REQUIRED >

<!ELEMENT license EMPTY >
<!ATTLIST license key CDATA #REQUIRED >

<!ELEMENT licenses ( license* ) >

<!ELEMENT nic EMPTY >
<!ATTLIST nic config CDATA #IMPLIED >
<!ATTLIST nic dnsip CDATA #IMPLIED >
<!ATTLIST nic duplex CDATA #IMPLIED >
<!ATTLIST nic ip CDATA #IMPLIED >
<!ATTLIST nic linkspeed CDATA #IMPLIED >
<!ATTLIST nic mac CDATA #IMPLIED >
<!ATTLIST nic name CDATA #REQUIRED >
<!ATTLIST nic status CDATA #REQUIRED >
<!ATTLIST nic subnet CDATA #IMPLIED >

<!ELEMENT nics ( nic* ) >

<!ELEMENT node ( nics, volumes ) >
<!ATTLIST node downtime CDATA #REQUIRED >
<!ATTLIST node hardwareversion CDATA #REQUIRED >
<!ATTLIST node name CDATA #REQUIRED >
<!ATTLIST node rail CDATA #REQUIRED >
<!ATTLIST node roles CDATA #REQUIRED >
<!ATTLIST node softwareversion CDATA #REQUIRED >
<!ATTLIST node status CDATA #REQUIRED >
<!ATTLIST node systemid CDATA #REQUIRED >
<!ATTLIST node totalrawcapacity CDATA #REQUIRED >
<!ATTLIST node usedrawcapacity CDATA #REQUIRED >

<!ELEMENT nodes ( node* ) >
```

```
<!ELEMENT pool ( applicationcontexts ) >
<!ATTLIST pool name CDATA #REQUIRED >
<!ATTLIST pool totalrawcapacity CDATA #REQUIRED >
<!ATTLIST pool usedrawcapacity CDATA #REQUIRED >

<!ELEMENT pools ( pool* ) >

<!ELEMENT rootswitch EMPTY >
<!ATTLIST rootswitch ip CDATA #REQUIRED >
<!ATTLIST rootswitch side CDATA #REQUIRED >
<!ATTLIST rootswitch status CDATA #REQUIRED >

<!ELEMENT rootswitches ( rootswitch* ) >

<!ELEMENT securityprofile ( capabilities ) >
<!ATTLIST securityprofile enabled CDATA #REQUIRED >
<!ATTLIST securityprofile name CDATA #REQUIRED >

<!ELEMENT servicecontentprotectiontransformation EMPTY >
<!ATTLIST servicecontentprotectiontransformation name CDATA #REQUIRED
  >
<!ATTLIST servicecontentprotectiontransformation scheme CDATA
  #REQUIRED >
<!ATTLIST servicecontentprotectiontransformation status CDATA
  #REQUIRED >
<!ATTLIST servicecontentprotectiontransformation threshold CDATA
  #REQUIRED >
<!ATTLIST servicecontentprotectiontransformation version CDATA
  #IMPLIED >

<!ELEMENT servicegarbagecollection EMPTY >
<!ATTLIST servicegarbagecollection name CDATA #REQUIRED >
<!ATTLIST servicegarbagecollection status CDATA #REQUIRED >
<!ATTLIST servicegarbagecollection version CDATA #IMPLIED >

<!ELEMENT serviceorganicregeneration EMPTY >
<!ATTLIST serviceorganicregeneration name CDATA #REQUIRED >
<!ATTLIST serviceorganicregeneration status CDATA #REQUIRED >
<!ATTLIST serviceorganicregeneration version CDATA #IMPLIED >

<!ELEMENT serviceperformanceregeneration EMPTY >
<!ATTLIST serviceperformanceregeneration name CDATA #REQUIRED >
<!ATTLIST serviceperformanceregeneration status CDATA #REQUIRED >
<!ATTLIST serviceperformanceregeneration version CDATA #IMPLIED >

<!ELEMENT servicequery EMPTY >
<!ATTLIST servicequery name NMTOKEN #REQUIRED >
<!ATTLIST servicequery status CDATA #REQUIRED >
<!ATTLIST servicequery version CDATA #IMPLIED >

<!ELEMENT servicereplication EMPTY >
<!ATTLIST servicereplication ip CDATA #REQUIRED >
```

```
<!ATTLIST servicereplication name NMTOKEN #REQUIRED >
<!ATTLIST servicereplication status CDATA #REQUIRED >
<!ATTLIST servicereplication version CDATA #IMPLIED >

<!ELEMENT servicerestore EMPTY >
<!ATTLIST servicerestore ip CDATA #REQUIRED >
<!ATTLIST servicerestore name NMTOKEN #REQUIRED >
<!ATTLIST servicerestore status CDATA #REQUIRED >
<!ATTLIST servicerestore version CDATA #IMPLIED >

<!ELEMENT services ( servicegarbagecollection*,
   servicecontentprotectiontransformation*, servicereplication*,
   servicerestore*, servicequery*, serviceorganicregeneration*,
   serviceperformanceregeneration*, serviceshredding*, servicesnmp* )
   >

<!ELEMENT serviceshredding EMPTY >
<!ATTLIST serviceshredding name NMTOKEN #REQUIRED >
<!ATTLIST serviceshredding status CDATA #REQUIRED >
<!ATTLIST serviceshredding version CDATA #IMPLIED >

<!ELEMENT servicesnmp EMPTY >
<!ATTLIST servicesnmp communityname CDATA #REQUIRED >
<!ATTLIST servicesnmp ip CDATA #REQUIRED >
<!ATTLIST servicesnmp name NMTOKEN #REQUIRED >
<!ATTLIST servicesnmp port CDATA #REQUIRED >
<!ATTLIST servicesnmp status CDATA #REQUIRED >
<!ATTLIST servicesnmp trapinterval CDATA #REQUIRED >
<!ATTLIST servicesnmp version CDATA #IMPLIED >

<!ELEMENT volume EMPTY >
<!ATTLIST volume index CDATA #REQUIRED >
<!ATTLIST volume status CDATA #REQUIRED >
<!ATTLIST volume totalrawcapacity CDATA #IMPLIED >

<!ELEMENT volumes ( volume* ) >
```

Sample Discovery

```
<?xml version="1.0" ?>
<!DOCTYPE discovery SYSTEM "discovery-1.1.dtd" >
<discovery>
  <cluster clusterid="2fae48a2-1dd2-11b2-9a8e-ae2a1b42afc5"
   compliancemode="Basic" contactemail="" contactname=""
   groomingvisit="" l
ocation="" name="" protectionschemes="R61,M2" serial="" serviceid=""
   serviceinfo="" siteid="" softwareversion="" availablerawcapacit
y="942660386816" offlinerawcapacity="0"
   totalrawcapacity="942660386816" usedrawcapacity="0">
    <cabinets>
```

```
      <cabinet id="1" availablerawcapacity="942660386816"
   offlinerawcapacity="0" totalrawcapacity="942660386816"
   usedrawcapacity="0"
>
        <ats powersource="-1" status="-1"/>
        <cubeswitches>
          <cubeswitch ip="10.255.1.61" mac="00:00:cd:03:20:74"
   description="Allied Telesyn AT-RP48 Rapier 48 version 2.2.2-12
   05-Mar
-2002" name="c001sw0" rail="0" serial="49906220" status="1"/>
          <cubeswitch ip="10.255.1.62" mac="00:00:cd:03:1f:24"
   description="Allied Telesyn AT-RP48 Rapier 48 version 2.2.2-10
   21-Dec
-2001" name="c001sw1" rail="1" serial="49906217" status="1"/>
        </cubeswitches>
        <nodes>
       <node downtime="-1" hardwareversion="118032076" name="c001n01"
   rail="1" roles="access" softwareversion="2.1.0.287-1715" st
atus="online" systemid="3644ea04-1dd2-11b2-b183-b3e636608d6d"
   totalrawcapacity="0" usedrawcapacity="0">
          <nics>
          <nic ip="10.255.1.1" mac="00:02:b3:5e:9d:a3" name="eth0"
   status="1"/>
          <nic ip="10.255.1.1" mac="00:02:b3:5e:9d:a4" name="eth1"
   status="1"/>
            <nic dnsip="152.62.69.47" ip="10.68.129.61"
   mac="00:e0:81:02:ae:64" config="D" duplex="full" linkspeed="100"
   name="eth
2" status="1" subnet="255.255.255.0"/>
          </nics>
          <volumes>
            <volume index="0" status="1"/>
            <volume index="1" status="1"/>
            <volume index="2" status="1"/>
            <volume index="3" status="1"/>
          </volumes>
        </node>
       <node downtime="-1" hardwareversion="118032076" name="c001n02"
   rail="0" roles="access" softwareversion="2.1.0.287-1715" st
atus="online" systemid="276e989a-1dd2-11b2-9a8e-ae2a1b42afc5"
   totalrawcapacity="0" usedrawcapacity="0">
          <nics>
          <nic ip="10.255.1.2" mac="00:02:b3:5f:c5:7a" name="eth0"
   status="1"/>
          <nic ip="10.255.1.2" mac="00:02:b3:5f:c5:7b" name="eth1"
   status="1"/>
            <nic dnsip="152.62.69.47" ip="10.68.129.62"
   mac="00:e0:81:02:8f:4e" config="D" duplex="full" linkspeed="100"
   name="eth
2" status="1" subnet="255.255.255.0"/>
          </nics>
          <volumes>
```

```
                <volume index="0" status="1"/>
                <volume index="1" status="1"/>
                <volume index="2" status="1"/>
                <volume index="3" status="1"/>
              </volumes>
            </node>
          <node downtime="-1" hardwareversion="118032076" name="c001n04"
   rail="0" roles="storage" softwareversion="2.1.0.287-1715" s
tatus="online" systemid="3e6cd318-1dd2-11b2-8516-a0a93ace538f"
   totalrawcapacity="145927176192" usedrawcapacity="0">
              <nics>
                <nic ip="10.255.1.4" name="eth0" status="1"/>
                <nic ip="10.255.1.4" name="eth1" status="1"/>
                <nic name="eth2" status="1"/>
              </nics>
              <volumes>
                <volume index="0" status="1"/>
                <volume index="1" status="1"/>
              </volumes>
            </node>
          <node downtime="-1" hardwareversion="118032076" name="c001n05"
   rail="1" roles="storage" softwareversion="2.1.0.287-1715" s
tatus="online" systemid="43737dbc-1dd2-11b2-aac6-c6edfea63736"
   totalrawcapacity="217717932032" usedrawcapacity="0">
              <nics>
                <nic ip="10.255.1.5" name="eth0" status="1"/>
                <nic ip="10.255.1.5" name="eth1" status="1"/>
                <nic name="eth2" status="1"/>
              </nics>
              <volumes>
                <volume index="0" status="1"/>
                <volume index="2" status="1"/>
                <volume index="3" status="1"/>
              </volumes>
            </node>
          <node downtime="-1" hardwareversion="118032076" name="c001n06"
   rail="0" roles="storage" softwareversion="2.1.0.287-1715" s
tatus="online" systemid="3ba37a9c-1dd2-11b2-aec0-c5883f6d2501"
   totalrawcapacity="289507639296" usedrawcapacity="0">
              <nics>
                <nic ip="10.255.1.6" name="eth0" status="1"/>
                <nic ip="10.255.1.6" name="eth1" status="1"/>
                <nic name="eth2" status="1"/>
              </nics>
              <volumes>
                <volume index="0" status="1"/>
                <volume index="1" status="1"/>
                <volume index="2" status="1"/>
                <volume index="3" status="1"/>
              </volumes>
            </node>
```

```
        <node downtime="-1" hardwareversion="118032076" name="c001n07"
    rail="1" roles="storage" softwareversion="2.1.0.287-1715" s
  tatus="online" systemid="4133cf0c-1dd2-11b2-8ebc-af799f89fd28"
    totalrawcapacity="289507639296" usedrawcapacity="0">
          <nics>
            <nic ip="10.255.1.7" name="eth0" status="1"/>
            <nic ip="10.255.1.7" name="eth1" status="1"/>
            <nic name="eth2" status="1"/>
          </nics>
          <volumes>
            <volume index="0" status="1"/>
            <volume index="1" status="1"/>
            <volume index="2" status="1"/>
            <volume index="3" status="1"/>
          </volumes>
        </node>
      </nodes>
    </cabinet>
  </cabinets>
  <rootswitches/>
  <services>
   <servicegarbagecollection name="Garbage Collection" status="0"/>
    <servicereplication ip="" name="Replication" status="0"/>
    <serviceshredding name="Shredding" status="0"/>
   <servicesnmp ip="" communityname="public" name="SNMP" port="162"
  status="0" trapinterval="60"/>
  </services>
  <pools>
    <pool name="default" totalrawcapacity="942660386816"
  usedrawcapacity="0">
      <applicationcontexts>
        <applicationcontext>
          <client/>
          <securityprofile enabled="true" name="anonymous">
            <capabilities>
              <capability enabled="true" name="purge"/>
              <capability enabled="true" name="write"/>
              <capability enabled="true" name="privileged-delete"/>
              <capability enabled="true" name="exist"/>
              <capability enabled="true" name="delete"/>
              <capability enabled="true" name="clip-enumeration"/>
              <capability enabled="true" name="monitor"/>
              <capability enabled="true" name="read"/>
            </capabilities>
          </securityprofile>
        </applicationcontext>
        <applicationcontext>
          <client/>
          <securityprofile enabled="true" name="top">
            <capabilities>
              <capability enabled="true" name="purge"/>
              <capability enabled="true" name="write"/>
```

```
                    <capability enabled="true" name="privileged-delete"/>
                    <capability enabled="true" name="exist"/>
                    <capability enabled="true" name="delete"/>
                    <capability enabled="true" name="clip-enumeration"/>
                    <capability enabled="true" name="monitor"/>
                    <capability enabled="true" name="read"/>
                  </capabilities>
                </securityprofile>
              </applicationcontext>
            </applicationcontexts>
          </pool>
        </pools>
        <licenses/>
      </cluster>
    </discovery>
```

Statistical Information

This section lists the syntax and a sample of the XML file that can be retrieved by `FPMonitor_GetAllStatistics`.

Syntax Statistics

```
<!ELEMENT statistics ( cluster?, nodes? ) >

<!ELEMENT cluster (stats?) >

<!ELEMENT nodes (node*) >

<!ELEMENT node (stats?) >
<!ATTLIST node name CDATA #IMPLIED >
<!ATTLIST node systemid CDATA #REQUIRED >
<!ATTLIST node type (access|storage|spare) #REQUIRED>

<!ELEMENT stats (stat*) >

<!ELEMENT stat EMPTY >
<!ATTLIST stat name CDATA #REQUIRED >
<!ATTLIST stat type (long|float|string) "string" >
<!ATTLIST stat value CDATA #REQUIRED >
```

Sample Statistics

```
<?xml version="1.0" ?>
<!DOCTYPE statistics SYSTEM "statistics-1.0.dtd" >
<statistics>
  <cluster>
    <stats>
      <stat name="bandwidth_replication_mb_15" type="float"
   value="0.0"/>
      <stat name="bandwidth_replication_mb_60" type="float"
   value="0.0"/>
      <stat name="bandwidth_replication_obj_1" type="float"
   value="0.0"/>
      <stat name="bandwidth_replication_obj_15" type="float"
   value="0.0"/>
      <stat name="bandwidth_replication_obj_60" type="float"
   value="0.0"/>
      <stat name="capacity_offline" type="long" value="0"/>
      <stat name="capacity_used" type="long" value="0"/>
      <stat name="replication_clips" type="long" value="0"/>
    </stats>
  </cluster>
```

```
  <nodes>
    <node name="c001n01"
   systemid="3644ea04-1dd2-11b2-b183-b3e636608d6d" type="access">
       <stats>
         <stat name="capacity_offline" type="long" value="0"/>
         <stat name="capacity_used" type="long" value="0"/>
       </stats>
    </node>
    <node name="c001n02"
   systemid="276e989a-1dd2-11b2-9a8e-ae2a1b42afc5" type="access">
       <stats>
         <stat name="capacity_offline" type="long" value="0"/>
         <stat name="capacity_used" type="long" value="0"/>
       </stats>
    </node>
    <node name="c001n04"
   systemid="3e6cd318-1dd2-11b2-8516-a0a93ace538f" type="storage">
       <stats>
         <stat name="capacity_offline" type="long" value="0"/>
         <stat name="capacity_used" type="long" value="0"/>
       </stats>
    </node>
    <node name="c001n05"
   systemid="43737dbc-1dd2-11b2-aac6-c6edfea63736" type="storage">
       <stats>
         <stat name="capacity_offline" type="long" value="0"/>
         <stat name="capacity_used" type="long" value="0"/>
       </stats>
    </node>
    <node name="c001n06"
   systemid="3ba37a9c-1dd2-11b2-aec0-c5883f6d2501" type="storage">
       <stats>
         <stat name="capacity_offline" type="long" value="0"/>
         <stat name="capacity_used" type="long" value="0"/>
       </stats>
    </node>
    <node name="c001n07"
   systemid="4133cf0c-1dd2-11b2-8ebc-af799f89fd28" type="storage">
       <stats>
         <stat name="capacity_offline" type="long" value="0"/>
         <stat name="capacity_used" type="long" value="0"/>
       </stats>
    </node>
  </nodes>
</statistics>
```

Alert Information

This section lists the syntax and a sample of the XML file that can be retrieved by `FPEventCallback_RegisterForAllEvents`.

Syntax Alert

```
<!ELEMENT alert (failure)>
<!ATTLIST alert type (degradation|improvement) #REQUIRED>

<!ELEMENT failure (node, device)>

<!ELEMENT node EMPTY>
<!ATTLIST node systemid CDATA #REQUIRED>

<!ELEMENT device EMPTY>
<!ATTLIST device type (node|disk|switch|rootswitch|nic|sdk)
   #REQUIRED>
<!ATTLIST device name CDATA #REQUIRED>
```

Sample Alert

```
<?xml version="1.0"?>
<!DOCTYPE alert SYSTEM "alert-1.0.dtd">

<alert type="degradation">
   <failure>
     <node systemid="[systemid of node]" />
     <device type="nic" name="[name of device]" />
   </failure>
</alert>
```

Sensors and Alerts

Centera sensors continually run on a cluster to monitor its state and raise alerts when appropriate. An alert is a message in XML format with information on the cluster's state to indicate a warning, error, critical situation, or notification. Refer to Table A-1, *Centera Sensor Based Alerts*, on page A-14 for a complete listing of all Centera alerts with their symptom code, error level, and a detailed description.

Centera alerts can be sent through:

- **ConnectEMC**: If ConnectEMC is enabled, alerts will automatically be sent to the EMC Customer Support Center, which decides if intervention by an EMC engineer is necessary. ConnectEMC uses an XML format for sending alert messages to EMC. The XML messages are encrypted. ConnectEMC sends email messages to the EMC Customer Support Center via the customer SMTP infrastructure or via a customer workstation with EMC OnAlert™ installed. The system administrator uses the CLI to configure ConnectEMC and to view the ConnectEMC configuration. Refer to the *Centera Online Help*, P/N 300-002-656, for more information on ConnectEMC-related CLI commands.
- **SNMP**: The Simple Network Management Protocol (SNMP) is an Internet-standard protocol for managing devices on IP networks. SNMP allows Centera to send alerts to storage management software such as the EMC ControlCenter family. The system administrator uses the CLI to configure SNMP and to view the current state of the SNMP configuration. Refer to the *Centera Online Help*, P/N 300-002-656, for more information on the SNMP-related CLI commands and SNMP details.
- **ConnectEMC Notification**:The system administrator can receive email notification with an HTML formatted copy of the alert message. Refer to the *Centera Online Help*, P/N 300-002-656, for more information.
- **EMC ControlCenter**: The EMC ControlCenter users can monitor one or more Centera clusters in their storage environment.
- **The Monitoring API (MoPI)**: The SDK has an ability to view raised alerts with the MoPI. Refer to *Monitoring Functions* on page 1-251 for more information on the MoPI functionality.

Table A-1 Centera Sensor Based Alerts

Symptom Code	Error Level	Sensor Name, Description and Resolution
1.1.1.1.01.01 1.1.1.1.01.02 1.1.1.1.01.03	Warning > 10000 Error > 25000 Critical > 50000	*ParkedReplicationsClipCount* The replication parking on 1 or more of the nodes in the cluster contains more than 10000/25000/50000 entries. This is typically due to connectivity issues between this cluster and the replica, or authorization and capability issues on the replica. In order to prevent the replication parking to increase, CentraStar has paused the replication process.
1.1.1.1.05.01 1.1.1.1.05.02 1.1.1.1.05.03	Warning > 10000 Error > 25000 Critical > 50000	*ParkedRestoreClipCount* The restore parking on 1 or more of the nodes in the cluster contains more than 10000/25000/50000 entries. This is typically due to connectivity issues between this cluster and the replica, or authorization and capability issues on the replica. In order to prevent the restore parking to increase, CentraStar has paused the restore process.
1.1.1.1.03.01 1.1.1.1.03.02 1.1.1.1.03.03 1.1.1.1.03.04 = -1	Warning >= 24 h Error >= 48 h Critical >= 120h	*ReplicationETA* The replication process on the cluster will probably take longer than 24/48/120/infinite hours to process the entire replication queue, taking into account write and replication rates over the past 24 hours. This might indicate a problem with replication or a write activity that is higher than average.
1.1.1.1.20.01	paused_clusterfull	*ReplicationSubstate* CentraStar has paused the replication process on the cluster because the replica cluster has not enough capacity to service the replication requests.

Table A-1 Centera Sensor Based Alerts (continued)

Symptom Code	Error Level	Sensor Name, Description and Resolution
1.1.1.1.20.02	paused_parking_overflow	*ReplicationSubstate* The replication parking on this cluster contains more than 10,000 entries on 1 or more nodes. This is typically due to connectivity issues between this cluster and the replica, or authorization and capability issues on the replica.
1.1.1.1.20.03	paused_no_capability	*ReplicationSubstate* The replica cluster has refused a replication request because the access profile used for replication does not have the correct capabilities.
1.1.1.1.20.04	paused_authentication_failure	*ReplicationSubstate* The replica cluster has refused a replication request because the access profile used for replication could not be authenticated.
1.1.1.1.20.05	paused_pool_not_found	*ReplicationSubstate* The target cluster has refused a replication request because the pool for the C-Clip being replicated was not found on the target cluster.
1.1.1.1.20.06	Warning	*ReplicationPaused* The replication has been paused by the user.
1.1.1.1.20.07	Warning	*ReplicationCancelled* The replication has been cancelled.
1.1.2.1.02.01	Critical < 0 (1=Basic, -2=CE (Governance), -3=CE+)	*ComplianceModeRevert* One or more nodes in the cluster do not share the cluster compliance settings. EMC Support has been notified of the problem and a case will automatically be opened.
1.1.3.1.01.01 1.1.3.1.01.02 1.1.3.1.01.03	Warning = 80% Error = 90% Critical = 100%	*PoolCloseToQuota* A pool has reached 80%, 90%, or 100% of its available quota.

Table A-1 Centera Sensor Based Alerts (continued)

Symptom Code	Error Level	Sensor Name, Description and Resolution
1.1.4.1.06.01	Error	*Restore Paused* The target cluster does not have enough capacity to process the restore request.
1.1.4.1.06.02	Error	*Restore Paused* The restore parking on the cluster contains more than 10,000 entries on one or more nodes.
1.1.4.1.06.03	Error	*Restore Paused* The target cluster has refused the restore request because the access profile used for restore does not have the correct capabilities.
1.1.4.1.06.04	Error	*Restore Paused* The access profile used for the restore could not be authenticated.
1.1.4.1.06.05	Error	*Restore Paused* The target cluster has refused the restore request because the pool for the C-Clip being restored was not found on the target cluster.
1.1.4.1.06.06	Warning	*Restore Paused* The restore has been paused by the user.
1.1.4.1.06.07	Warning	The restore has been cancelled.
2.1.2.1.01.01	Error = INCOMPLETE	*ClusterClipIntegrity* CentraStar has determined that 1 or more C-Clips have become unavailable due to a combination of disks and/or nodes being offline. EMC Support has been notified of the problem and a case will automatically be opened.

Table A-1 Centera Sensor Based Alerts (continued)

Symptom Code	Error Level	Sensor Name, Description and Resolution
3.1.3.1.01.01 3.1.3.1.01.02	Warning = 1 Error > 1	*InternalNICFailureCount* CentraStar has determined that 1 or more NICs used by the internal Centera network is not functioning correctly. EMC Support has been notified of the problem and a case will automatically be opened.
3.1.3.1.02.01	Error > 1	*ExternalNICFailureCount* CentraStar has determined that 1 or more NICs used for data access by your application is not functioning correctly. This will reduce bandwidth available for data access by your application. EMC Support has been notified of the problem and a case will automatically be opened.
3.1.4.1.01.01 3.1.4.1.01.02	Error = 1 Critical > 1	*CubeSwitchFailureCount* CentraStar has determined that 1 or more cube switches in the cluster is not functioning correctly. This will reduce the internal bandwidth of the cluster and could result in data becoming unavailable. EMC Support has been notified of the problem and a case will automatically be opened.
3.1.4.1.02.01	Warning	*Root Switch Failure* CentraStar has determined that 1 or more root switches has malfunctioned. When a switch fails in a multi-rack configuration, two alert messages are sent: One indicates the switch failure; the other indicates a failure on the root switch connected to it. EMC Support has been notified of the problem and a case will automatically be opened.

Table A-1 Centera Sensor Based Alerts (continued)

Symptom Code	Error Level	Sensor Name, Description and Resolution
3.1.5.1.01.01	Warning	*Disk Offline* A hardware or software failure has caused a disk to go offine. EMC Support has been notified of the problem and a case will automatically be opened
3.1.6.1.01.01	Warning	*Node Offline* A hardware or software failure has caused a node to go offine. EMC Support has been notified of the problem and a case will automatically be opened
4.1.1.1.02.01	Error > 88%	*VarPartitionUsedSpace* CentraStar has determined that 1 or more nodes in the cluster with an internal partition used by CentraStar have limited capacity. EMC Support has been notified of the problem and a case will automatically be opened.
5.2.2.1.03.01 5.2.2.1.03.02 5.2.2.1.03.03	Warning < 20% Error < 10% Critical <= 5%	*AvailableCapacityHardStop Percent* The available capacity on the cluster is less than 20%/10%/5% of the total raw capacity. As there is no regeneration buffer defined, it will soon not be possible to write to the cluster anymore and disk and node regenerations will not be able to complete.
5.2.2.1.04.01 5.2.2.1.04.02 5.2.2.1.04.03	Warning > 80% Error > 90% Critical >= 95%	*FreeObjectPercent* The object count on the cluster is more than 80/90/95% of the allowed object count. It will soon not be possible to write to the cluster anymore and disk and node regenerations will not be able to complete.

Table A-1 Centera Sensor Based Alerts (continued)

Symptom Code	Error Level	Sensor Name, Description and Resolution
5.2.2.1.01.01 5.2.2.1.01.02 5.2.2.1.01.03 5.2.2.1.01.04	Notification < 125 % Warning < 100 % Error < 50 % Critical < 25%	*RegenerationBufferAlert Percent* The available capacity on the cluster is 25% lower than/less than/less than 50% of/25% of the configured regeneration buffer. As the policy is set to `alert only` it is still possible to write to the cluster.
5.2.2.1.02.01 5.2.2.1.02.02 5.2.2.1.02.03	Warning < 150% Error = 125% Critical = configured buffer	*Available Capacity Close to Regeneration Hard Stop* The available capacity on at least one cube in the cluster is 150% lower than the configured buffer,125% of the configured buffer, or equal to the configured buffer. For the Error and Critical levels, EMC Support has been notified of the problem and a case will automatically be opened.
5.2.2.1.01 5.2.2.1.02 5.2.2.1.03	Warning < 150% Error < 125% Critical <= 100%	*RegenerationBufferHardStopPercent* The available capacity on the cluster is 50%/25% lower than/less than the configured regeneration buffer. As the policy is set to `hardstop` it will soon/is not possible to write to the cluster anymore. Disk and node regenerations will still be able to use the available capacity in the regeneration buffer.
4.1.1.1.01.01 4.1.1.1.01.02	Warning > 66 °C Error > 86 °C	*CPUTemperature* CentraStar has determined that 1 or more nodes in the cluster has a running temperature higher than 66 °C/86 °C. This will reduce the performance of the affected nodes and if the situation persists, the node will go offline. EMC Support has been notified of the problem and a case will automatically be opened.

Table A-1 Centera Sensor Based Alerts (continued)

Symptom Code	Error Level	Sensor Name, Description and Resolution
4.1.1.1.02.01	Error>88% capacity used	*Available System Capacity* One or more nodes with an internal partition has used more than 88% of its capacity. EMC Support has been notified of the problem and a case will be opened automatically.
3.1.6.01.01[a]	Warning = 1	*NodesOffline* There are 1 or more nodes offline in the cluster. EMC Support has been notified of the problem and a case will automatically be opened.
3.1.5.1.01[a]	Warning = 1	*DiskOffline* There are 1 or more disks offline in the cluster. EMC Support has been notified of the problem and a case will automatically be opened.
3.1.4.1.02[a]	Warning > 1	*RootSwitchFailures* There is 1 root switch in the cluster not functioning correctly. EMC Support has been notified of the problem and a case will automatically be opened.

a.This is a hardware alert. Hardware alerts are not raised by a Centera sensor.

Note: Do not contact EMC if ConnectEMC is enabled. The EMC Customer Support Center will automatically be notified by ConnectEMC.

Sample Alert

Alert messages sent via ConnectEMC to the customer are in HTML format. This is a sample HTML alert message:

```
Centera Alert Report

Version 1.1

Alert Identification

Type: Hardware
Symptom Code: 3.1.3.1.01.01
Format Version: 1.1
Format DTD: alert_email-1.1.dtd
Creation Date and Time: 02-01-2005 07:54:02.562 UTC

Cluster Identification

Name: cluster138
Serial Number: APM25431700200
Revision Number: no data available
Cluster ID: c3cd501c-1xx1-11b2-b513-d826a75388b0
Cluster Domain: company.com

Alert Description

Alert Level: WARNING
Alert Description:
   $Hardware.Main.NIC.Main.InternalNICFailureCount@$CLUST
   ER:eth0
Component: node
Component ID: c001n05
Sub-Component: nic
Sub-Component ID: eth0
```

B

Deprecated Functions

This appendix lists deprecated Centera API functions and options. Deprecated functions and options are not supported; client applications should not use them. If you need information about these functions or options—for example, to interpret existing code—refer to the Javadoc or Doxygen documentation included with this release.

The sections in this appendix are:

Deprecated Functions

Table B-1 lists deprecated Centera API functions.

Table B-1 Deprecated Functions

Function Name	Deprecated Release	Comments
FPTime_LongToString	3.1	Use the FPTime_SecondsToString or FPTime_MillisecondsToString function. Refer to *Time Functions* on page 1-267.
FPTime_StringToLong	3.1	Use the FPTime_StringToSeconds or FPTime_StringToMilliseconds function. Refer to *Time Functions* on page 1-267.
FPClip_Purge	2.3	Use the FP_OPTION_DELETE_PRIVILEGED option of FPClip_AuditedDelete() (Page 1-48) to delete content before the retention period has expired.
FPTag_BlobPurge	2.3	Garbage collection purges unreferenced blobs automatically.
FPQuery_Open	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to *Query Functions* on page 1-223.
FPQuery_GetPoolRef	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to *Query Functions* on page 1-223.
FPQuery_FetchResult	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to *Query Functions* on page 1-223.
FPQuery_Close	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to *Query Functions* on page 1-223.
FPStream_CreateWithBuffer	1.1	Use FPStream_CreateBufferForInput() (Page 1-194) and FPStream_CreateBufferForOutput() (Page 1-195).
FPStream_CreateWithFile	1.1	Use FPStream_CreateFileForInput() (Page 1-196) and FPStream_CreateFileForOutput() (Page 1-198).
FPStream_Read	1.1	Use the generic stream facility. Refer to *Stream Functions* on page 1-189.
FPStream_Write	1.1	Use the generic stream facility. Refer to *Stream Functions* on page 1-189.
FPStream_SetMarker	1.1	Use the generic stream facility. Refer to *Stream Functions* on page 1-189.
FPStream_GetMarker	1.1	Use the generic stream facility. Refer to *Stream Functions* on page 1-189.

Deprecated Options

Table B-2 lists deprecated Centera API options.

Table B-2 Deprecated Options

Option Name	Deprecated Release	Comments
FP_OPTION_ENABLE_DUPLICATE_DETECTION	2.1	Was used by FPTag_BlobWrite(). Duplicate detection is now always enabled.
FP_OPTION_CALCID_NOCHECK	2.1	Was used by FPTag_BlobWrite(), FPTag_BlobRead(), and FPTag_BlobReadPartial(). There is now always end-to-end checking of the Content Address on the client and the server to verify the content.

Glossary

This glossary contains terms used in this manual that are related to disk storage subsystems.

A

Access Node See *Node with the Access Role*.

Application Program(ming) Interface (API) A set of function calls that enables communication between applications or between an application and an operating system.

Automatic (AC) Transfer Switch (ATS) An AC power transfer switch. Its basic function is to deliver output power from one of two customer facility AC sources. It guarantees that the cluster will continue to function if a power failure occurs on one of the power sources by automatically switching to the secondary source.

B

Blob The Distinct Bit Sequence (DBS) of user data. The DBS represents the actual content of a file and is independent of the filename and physical location.

Note: Do not confuse this term with the term "Binary Large Object" that exists in the database sector.

C

C-Clip A package containing the user's data and associated metadata. When a user presents a file to the Centera system, the system calculates a unique Content Address (CA) for the data and then stores the file. The system also creates a separate XML file containing the CA of the user's file and application-specific metadata. Both the XML file and the user's data are stored in the C-Clip.

C-Clip Descriptor File (CDF) The additional XML file that the system creates when making a C-Clip. This file includes the Content Addresses for all referenced blobs and associated metadata.

C-Clip ID The Content Address that the system returns to the client. It is also referred to as a C-Clip handle or C-Clip reference.

Cluster One or more cabinets on which the nodes are clustered. Clustered nodes are automatically aware of nodes that attach to and detach from the cluster.

Cluster Time The synchronized time of all the nodes within a cluster.

Command Line Interface (CLI) A set of predefined commands that you can enter via a command line. The Centera CLI allows a user to manage a cluster and monitor its performance.

Content Address (CA) An identifier that uniquely addresses the content of a file and not its location. Unlike location-based addresses, Content Addresses are inherently stable and, once calculated, they never change and always refer to the same content.

Content Address Resolution The process of discovering the IP address of a node containing a blob with a given Content Address.

Content Address Verification The process of checking data integrity by comparing the CA calculations that are made on the application server (optional) and the nodes that store the data.

Content Addressed Storage (CAS) The generic term for a Centera cluster and its software. In the same way that a Symmetrix is considered a SAN device, a cluster is considered a CAS device.

Content Protection Mirrored (CPM) The content protection scheme where each stored object is copied to another node on a Centera cluster to ensure data redundancy.

Content Protection Parity (CPP) The content protection scheme where each object is fragmented into several segments that are stored on separate nodes with a parity segment to ensure data redundancy.

Cube A collection of 8, 16, 24, or 32 nodes and two cube switches, forming the basic building block for a cluster.

D

Distinct Bit Sequence (DBS) The actual content of a file independent of the filename and physical location. Every file consists of a unique sequence of bits and bytes. The DBS of a user's file is referred to as a blob in the Centera system.

Dynamic Host Configuration Protocol (DHCP) An internet protocol used to assign IP addresses to individual workstations and peripherals in a LAN.

E

End-to-end checking The process of verifying data integrity from the application end down to the second node with the storage role. See also Content Address Verification.

Extensible Markup Language (XML) A flexible way to create common information formats and share both the format and the data on the World Wide Web, intranets, and elsewhere. Refer to http://www.xml.com for more information.

F

Failover Commonly confused with failure. It actually means that a failure is transparent to the user because the system will "fail over" to another process to ensure completion of the task; for example, if a disk fails, then the system will automatically find another one to use instead.

I

Input parameter The required or optional information that has to be supplied to a function.

L

Load balancing The process of selecting the least-loaded node for communication. Load balancing is provided in two ways: first, an application server can connect to the cluster by selecting the least-loaded node with the access role; second, this node selects the least loaded node with the storage role to read or write data.

Local Area Network (LAN) A set of linked computers and peripherals in a restricted area such as a building or company.

M

Message Digest 5 (MD5) A unique 128-bit number that is calculated by the Message Digest 5-hash algorithm from the sequence of bits (DBS) that constitute the content of a file. If a single byte changes in the file then any resulting MD5 will be different.

Mirror team A logical organization of a number of nodes that always mirror each other.

MultiCast Protocol (MCP) A network protocol used for communication between a single sender and multiple receivers.

N

Node Logically, a network entity that is uniquely identified through a system ID, IP address, and port. Physically, a node is a computer system that is part of the Centera cluster.

Node with the Access Role The nodes in a cluster that communicate with the outside world. They must have public IP addresses. For clusters with CentraStar 2.3 and lower this was referred to as Access Node.

Node with the Storage Role The nodes in a cluster that store data. For clusters with CentraStar 2.3 and lower this was referred to as Storage Node.

O

Output parameter The information that a function returns to the application that called the function.

P

Pool A set of separate clusters that are linked together to constitute one Content Addressed Storage device.

Pool Transport Protocol (PTP) A further evolution of the UniCast Protocol (UCP) used for communication over the Internet between the application server and a node with the access role.

Probing A process where the application server requests information from the cluster to determine if it should start a PTP session.

R

Redundancy A process where data objects are duplicated or encoded such that the data can be recovered given any single failure. Refer to *Content Protection Mirrored (CPM)*, *Content Protection Parity (CPP)*, and, *Replication* for specific redundancy schemes used in Centera.

Regeneration The process of creating a data copy if a mirror copy or fragmented segment of that data is no longer available.

Relaying A way of streaming data directly from a node with the storage role over a node with the access role to the application server in case the access cache does not contain the requested data.

Replication The process of copying a blob to another cluster. This complements Content Protection Mirrored and Content Protection Parity. If a problem renders an entire cluster inoperable, then the replica cluster can keep the system running while the problem is fixed.

Retention Period The time that a C-Clip and the underlying blobs have to be stored before the application is allowed to delete them.

Return value The outcome of a function that the system returns to the application calling the function.

S

Segmentation The process of splitting very large files or streams into smaller chunks before storing them. Segmentation is an invisible client-side feature and supports storage of very large files such as rich multimedia.

Spare node A node without role assignment. This node can become a node with the access and/or storage role.

Storage Node See *Node with the Storage Role*.

Stream Generalized input/output channels that provide a way to handle incoming and outgoing data without having to know where that data comes from or goes to.

T

Time to First Byte (TTFB) The time between the request to the system to retrieve a C-Clip and the retrieval of the first byte of the blob.

U

UniCast Protocol (UCP) A network protocol used for communication between multiple senders and one receiver.

User Datagram Protocol (UDP) A standard Internet protocol used for the transport of data.

W

Wide Area Network (WAN) A set of linked computers and peripherals that are not in one restricted area but that can be located all over the world.

Write Once Read Many (WORM) A technique that stores data that will be accessed regularly, for example, a tape device.

Index

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X