EMC® Centera®

SDK Version 3.2

API Reference Guide

P/N 069001185
REV A08

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

Published September, 2007

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.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com.

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

Third-Party License Agreements

EMC Centera Software Development Kit

The EMC 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.

Use of GPL

The EMC® version of Linux® , used as the operating system on the EMC Centera server, is a derivative of Red Hat® and SuSE Linux. The operating system is copyrighted and licensed pursuant to the GNU General Public License (GPL), a copy of which can be found in the accompanying documentation. Please read the GPL carefully, because by using the Linux operating system on the EMC Centera server, you agree to the terms and conditions listed therein.

SKINLF

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

Bouncy Castle

The Bouncy Castle Crypto package is Copyright © 2000 of The Legion Of The Bouncy Castle (http://www.bouncycastle.org).

RSA Data Security

Copyright © 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

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.

ICU License (IBM International Component on Unicode library)

Apache Software Foundation

ReiserFS

how best to address the fair crediting issue in the next GPL version.)

MIT XML Parser

MIT XML Parser software is included. This software includes Copyright (c) 2002,2003, Stefan Haustein, Oberhausen, Rhld., Germany

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 included in all copies or substantial portions of the Software.

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. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

Chapter 5 Blob Handling Functions

Chapter 6 Retention Class Functions

Chapter 7 Stream Functions

Chapter 10 Time Functions

Chapter 11 Logging Functions

Tables

Preface

As part of an effort to improve and enhance the performance and capabilities of its product line, EMC from time to time releases revisions of its hardware and software. Therefore, some functions described in this document may not be supported by all revisions of the software or hardware currently in use. For the most up-to-date information on product features, refer to your product release notes.

If a product does not function properly or does not function as described in this document, please contact your EMC representative.

Audience

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

Readers of this guide are expected to be familiar with the following topics:

- Content Addressed Storage (CAS)
- EMC Centera architecture and features

Related documentation

Related documents include:

- *EMC Centera Quick Start Guide*
- *EMC Centera Online Help*
- *EMC Centera Programmer's Guide*
- *EMC Centera SDK Release Notes* (for supported platforms)

Conventions used in this document

EMC uses the following conventions for special notices.

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

CAUTION

A caution contains information essential to avoid data loss or damage to the system or equipment.

IMPORTANT

An important notice contains information essential to operation of the software.

WARNING

A warning contains information essential to avoid a hazard that can cause severe personal injury, death, or substantial property damage if you ignore the warning.

DANGER

A danger notice contains information essential to avoid a hazard that will cause severe personal injury, death, or substantial property damage if you ignore the message.

Typographical conventions

EMC uses the following type style conventions in this document:

Normal	Used in running (nonprocedural) text for: • Names of interface elements (such as names of windows, dialog boxes, buttons, fields, and menus) • Names of resources, attributes, pools, Boolean expressions, buttons, DQL statements, keywords, clauses, environment variables, filenames, functions, utilities • URLs, pathnames, filenames, directory names, computer names, links, groups, service keys, file systems, notifications
Bold:	Used in running (nonprocedural) text for: • Names of commands, daemons, options, programs, processes, services, applications, utilities, kernels, notifications, system call, man pages Used in procedures for: • Names of interface elements (such as names of windows, dialog boxes, buttons, fields, and menus) • What user specifically selects, clicks, presses, or types

Italic:	Used in all text (including procedures) for: • Full titles of publications referenced in text • Emphasis (for example a new term) • Variables
`Courier:`	Used for: • System output, such as an error message or script • URLs, complete paths, filenames, prompts, and syntax when shown outside of running text
`Courier bold:`	Used for: • Specific user input (such as commands)
`Courier italic:`	Used in procedures for: • Variables on command line • User input variables
< >	Angle brackets enclose parameter or variable values supplied by the user
[ ]	Square brackets enclose optional values
|	Vertical bar indicates alternate selections - the bar means “or”
{ }	Braces indicate content that you must specify (that is, x or y or z)
...	Ellipses indicate nonessential information omitted from the example

Where to get help

EMC support, product, and licensing information can be obtained as follows.

Product information — For documentation, release notes, software updates, or for information about EMC products, licensing, and service, go to the EMC Powerlink website (registration required) at:

`http://Powerlink.EMC.com`

Technical support — For technical support, go to EMC Customer Service on Powerlink. To open a service request through Powerlink, you must have a valid support agreement. Please contact your EMC sales representative for details about obtaining a valid support agreement or to answer any questions about your account.

Your comments

Your suggestions will help us continue to improve the accuracy, organization, and overall quality of the user publications. Please send your opinion of this document to:

`techpub_comments@EMC.com`

1 Introduction

This chapter introduces the API function types and syntax conventions that comprise the EMC Centera Access C API. It also includes detailed server and SDK specifications.

The main sections in this chapter are:

Function syntax

This section details the syntax of EMC® 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.

Table 1 on page 20 shows the API functions categorized by function class:

Table 1 API function types

Function class	Function name prefix
Pool functions	`FPPool`
C-Clip functions	`FPClip` or `FPClipID`
Tag functions	`FPTag`
Retention Class functions	`FPRetentionClassContext` or `FPRetentionClass`
Stream functions	`FPStream`
Query functions • Pool query functions • Query result functions	`FPQueryExpression` • `FPPoolQuery` • `FPQueryResult`
Monitoring functions	`FPMonitor` or `FPEventCallback`
Time format functions	`FPTime`
Logging functions	`FPLogging` or `FPLogState`

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 26.

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 EMC 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 EMC 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 *EMC Centera Programmer's Guide* for more information.

Table 2 on page 22 lists the function variants.

Table 2 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: EMC 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`.

Table 3 on page 23 lists the input and output units for API calls that deal with string encoding.

Table 3 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`
- `FPLogging_Log`
- `FPLogging_OpenLogState`
- `FPLogState_SetLogPath`
- `FPLogState_Save`
- `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`
- `FPStream_CreatePartialFileForInput`
- `FPStream_CreatePartialFileForOutput`
- `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

Table 4 on page 26 lists the data types used by the EMC Centera API:

Table 4 EMC Centera 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	The 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 5 on page 27.
FPPoolInfo	The structure that specifies pool information. The application should not deallocate or modify the pointer member variables. The FPPoolInfo structure is detailed in Table 6 on page 27.
FPStreamInfo	The stream structure that passes information to and from callback functions. The FPStreamInfo structure is detailed in Table 7 on page 27.

Table 5 FPErrorInfo structure

Errorinfo type	Definition
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 (cluster error) • FP_CLIENT_ERRORCLASS (client application error, probably due to coding error or wrong usage)

Table 6 FPPoolInfo structure

PoolInfo type	Definition
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 7 FPStreamInfo structure (1 of 2)

StreamInfo type	Definition
short mVersion	The current version of FPStreamInfo.
void *mUserData	Application-specific data, untouched by the SDK.

Table 7 FPStreamInfo structure (2 of 2)

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.

SDK specifications

This section lists the specifications that apply to CentraStar® and SDK operations.

Note: Contact your EMC representative if there is a need to exceed any of the product specifications listed in the following tables.

Table 8 on page 29 shows the general C-Clip specifications.

Table 8 Content specifications (1 of 2)

Specification	Value	Note
Maximum number of tags per CDF	131072 or 100MB, whichever is larger	**CentraStar**: The number of tags cannot exceed 131072. **SDK**: The maximum is limited to how many tags can fit into 100MB (max CDF size).
Maximum CDF size	100MB	The maximum allowed segment size of 100MB.
Maximum blob size	100GB	The EMC-recommended limit.
Maximum size of an embedded blob	100KB (default)	The maximum by default. An application can change this default up to a maximum of 1MB. **Note:** Exceeding 100KB can have a negative impact on performance.
Maximum size of internal CDF buffer	10MB	The maximum size of data stored in memory before it is written to disk.
Maximum size of tag name	1KB (default)	The maximum by default. An application can change this default up to a maximum of 1MB. **Note:** Exceeding 100KB can have a negative impact on performance.

Table 8 Content specifications (2 of 2)

Specification	Value	Note
Maximum size of attribute name	1KB (default)	The maximum by default. An application can change this default up to a maximum of 1MB. **Note:** Exceeding 100KB can have a negative impact on performance.
Maximum size of attribute value	100KB (default)	The maximum by default. An application can change this default up to a maximum of 1MB. **Note:** Exceeding 100KB can have a negative impact on performance.
Maximum number of attributes in a tag	256 or 100MB, whichever is larger	**CentraStar**: The number of attributes cannot exceed 256. **SDK**: The maximum is limited to how many attributes can fit into 100MB (max CDF size).
Maximum number of nested tags in CDF	256 or 100MB, whichever is larger	**CentraStar**: The number of tags cannot exceed 256. **SDK**: The maximum is limited to how many nested tags can fit into 100MB (max CDF size).
Maximum size of Litigation Hold ID	64 characters	
Maximum Litigation (Retention) Hold IDs per C-Clip	100 total	The maximum is limited to the number of litigation holds ever created for a C-Clip (including deleted ones in the count).

Table 9 on page 31 shows the specifications for SDK operations.

Table 9 SDK pool operations

Specification	Value	Note
Supported maximum number of threads per access role	30	The maximum set by the server. For nodes that also have the storage role, the maximum is 15. **Note:** If writing large files greater than 2MB, EMC recommends a maximum of 10 threads per access role.
Maximum number of parallel queries	10	
Maximum pool open transactions per minute	5	The maximum set by the server.
Maximum number of connections to a pool	999	The default is 100.
Maximum number of SDK-provided sockets	999	The default is 100.
Stream memory buffer	VALUE > 0	The VALUE must be greater than 0, where VALUE is the OS minimum and is application-specified.
Maximum prefetch buffer	1MB	

Table 10 on page 32 shows the general cluster specifications.

Table 10 Cluster specifications

Specification	Value	Note
Maximum number of pools	100	The following pool names are reserved: • cluster • default • AuditArchive • ConsoleArchive • SystemArchive
Maximum number of profiles	100	The following profile names are reserved: • anonymous • admin • console • profile names beginning with emc
Maximum number of retention classes	1000	
Maximum number of nodes	128	
Maximum number of nodes with the access role	8 per rack	This number may not exceed half the number of nodes in a rack.
Minimum number of nodes with the storage role	4	
Maximum clusters per domain	8	

Table 11 on page 32 shows the specifications based on the node model and those that apply to newly installed nodes without data. The capacity figures are all in base2 notation (1KB = 1024 bytes) except for disk size, which is in base10:

Table 11 Node specifications (1 of 2)

Specification	Gen2	Gen3	Gen4	Gen4	Gen4LP
Disk size	250GB	320GB	320GB	500GB	750GB
Number of disks	4	4	4	4	4
Raw capacity	934.8GB	1206.8GB	1862.6GB	1862.6GB	2794.6GB

Table 11 **Node specifications (2 of 2)**

Specification	Gen2	Gen3	Gen4	Gen4	Gen4LP
System resources	6.8GB	6.8GB	10.8GB	10.8GB	10.8GB
Free capacity	927.8GB	1199.8GB	1195.6GB	1851.6GB	2783.3GB
Maximum objects	30M	30M	50M	50M	50M
Number of nodes in cube	8-32	8-32	4-16	4-16	4-16
Number of cubes in cluster	1-4	1-4	1-8	1-8	1-8

Table 12 on page 33 shows the general replication specifications.

Table 12 **Replication specifications**

Specification	Value	Note
Maximum number of incoming replication clusters	3	A maximum of 3 clusters can replicate to the same target cluster.
Maximum number of outgoing replication clusters	1	A source cluster can replicate to only 1 target cluster.
Maximum number of characters for the replica cluster's string of IP addresses	256	
Maximum number of replication addresses	4	

EMC Centera is designed to run the same CentraStar version on replicating clusters. To support CentraStar upgrade procedures and restore use cases, EMC Centera supports replication and restore between different CentraStar versions, as shown in Table 13 on page 34.

Table 13 Replication and restore compatibility matrix

Source/Target	3.1.3	3.1.2	3.1	3.0	2.4
3.1.3	R/r	R/r	R/c	R/c (2)	R/c (2)
3.1.2	R/r	R/r	R/c	R/c (2)	R/c (2)
3.1	R/c	R/c	R/c	R/c (2)	R/c (2)
3.0	R/c	R/c	R/c	R/c	R/c
2.4	R/c (1)	R/c (1)	R/c (1)	R/c (1)	R/c
R = Supports replication					
c = Supports restore					
C = CSGReplication should be used for replication and/or restore.					
(1) Data will end up in a pool, per current home pool assignments and active mappings.					
(2) Does not support replication of event-based retention and litigation hold. If attempted, the process is automatically paused.					

For service packs not listed in this matrix, refer to the last major release that preceded the service pack (for example, 3.1 for 3.1.1).

Table 14 on page 34 shows the general network specifications.

Table 14 Network specifications

Specification	Value	Note
Maximum number of networks	1	All nodes with the access role must connect to the same subnet.
NAT support between application server and cluster	Yes	
NAT support between clusters	No	
Support for static and dynamic VLAN	Yes	
VLAN tagging support	No	

2

Pool Functions

This chapter describes the `FPPool` API functions.

The main section in this chapter is:

Pool functions

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

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.

Here is the list of SDK pool functions:

- `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 22.

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 *EMC Centera Programmer's Guide* 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 15 on page 39 for the capability names, attribute names, and attribute values that represent the capabilities.

Table 15 Capability names and attributes (1 of 4)

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 15 Capability names and attributes (2 of 4)

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 15 Capability names and attributes (3 of 4)

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 CentraStar 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 CentraStar 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 CentraStar 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 CentraStar v3.1.

Table 15 Capability names and attributes (4 of 4)

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 *EMC Centera Programmer's Guide*.

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 72.

Error handling

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

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

Note: This call fails from the SDK side if the specified pool was not previously opened, if the Profile Clip was not previously set by a call to `FPPool_SetClipID`, or if the Profile Clip was deleted.

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 22.

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=128;
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 22.

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

Parameters `const char *inOptionName`

Refer to "FPPool_SetGlobalOption" on page 61 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 SDK may return `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 22.

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 58 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. "Error codes" on page 320 contains 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 5 on page 27 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 5 on page 27 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 EMC 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 *EMC Centera Online Help* 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 *EMC Centera Programmer's Guide* 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 22.

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 *EMC Centera Programmer's Guide* for more information. You can change the default open strategy; refer to "FPPool_SetGlobalOption" on page 61.

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 *EMC Centera Programmer's Guide* for more information.

To free associated resources, be sure to close all pool connections "FPPool_Close" on page 37 that are no longer needed.

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 *EMC Centera Programmer's Guide*.

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 *EMC Centera Programmer's Guide* 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 22.

Description

This function stores an application's name and version on the EMC Centera log. This API call allows for improvments in the application and customer support of integrated applications. All EMC Centera Proven certifications must use application registration for v3.1 or higher.

Note: To ensure that this application information is written to an EMC Centera log, you must call `FPPool_RegisterApplication` before calling `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 *EMC Centera Programmer's Guide*.

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 22.

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.

  Note: For available options and values, see "Global options" on page 62.

- `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)

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 EMC Centera server (not the client) to calculate the content address of the blob data on the server.
- FP_OPTION_ENABLE_DUPLICATE_DETECTION — Enables the SDK to detect whether a duplicate blob with the same content address exists in the cluster.
- 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 *EMC Centera Programmer's Guide* 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 EMC 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 37 and the *EMC Centera Programmer's Guide* 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 60 seconds (1 minute). 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 *EMC Centera Programmer's Guide* 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 EMC Centera operation. Table 16 on page 64 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, as described in "FPPool_SetIntOption" on page 68.

Table 16 Multicluster failover strategy options and values (1 of 3)

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.

Table 16 Multicluster failover strategy options and values (2 of 3)

Option: Value	Description
FP_REPLICATION_STRATEGY	Content is written to all eligible clusters synchronously before the operation completes and the Content Address is returned. Consider the following behavior: • 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.
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_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_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_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.

Table 16 Multicluster failover strategy options and values (3 of 3)

Option: Value	Description
FP_OPTION_MULTICLUSTER_QUERY_STRATEGY:	The multicluster failover strategy for query operations: FPPoolQuery_Open().
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, you can specify which cluster types participate in that strategy for each operation. Refer to Table 16 on page 64. 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. For more information, see "FPPool_Close" on page 37.

  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_SetIntOption" on page 68.

- `FP_OPTION_STREAM_STRICT_MODE` — The Boolean value for whether strict mode is enabled or disabled for field validation of the `FPStreamInfo` structure after every callback function. If true and logging is enabled and an error occurs, the SDK throws the `FP_STREAM_VALIDATION_ERR` error and generates the appropriate logs. If set to false and the same situation occurs, only warnings are logged. The default is set to true. For more information, see "FPStream_CreateGenericStream" on page 208.

  Note: If setting FP_OPTION_STREAM_STRICT_MODE as an environment variable, you can disable it by setting it to a non-zero value.

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 22.

Description

This function sets the options for the given pool. To change global pool settings, refer to "FPPool_SetGlobalOption" on page 61.

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. This value must be greater than 0.
  - `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 2-61). 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 *EMC Centera Programmer's Guide* 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 168.
- 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)

3

Clip Functions

This chapter describes the various types of FPClip functions.

The main sections in this chapter are:

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 FPClip 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 22.

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 38 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 94 and "FPClip_SetRetentionClass" on page 91 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 *EMC Centera Programmer's Guide* for more

information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 *EMC Centera Programmer's Guide* 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 238. 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 22.

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 38 for more information on the server capabilities, and to "FPClip_SetRetentionPeriod" on page 94 and "FPClip_SetRetentionClass" on page 91 for more information on retention periods.

Delete operations do not fail over by default in multicluster environments. Refer to the *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 73.

A C-Clip is deleted only 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 *EMC Centera Programmer's Guide* 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 *EMC Centera Online Help*.

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 238.

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 97 or "FPClip_TriggerEBREvent" on page 96.

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 EMC Centera. Writing a modified C-Clip creates a new C-Clip with a new clip ID, and does not change the existing C-Clip.

Note: 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 38.)

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

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 99, "FPClip_TriggerEBREventWithClass" on page 97, and "FPClip_TriggerEBREvent" on page 96.

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 EMC Centera. Writing a modified C-Clip creates a new C-Clip with a new clip ID, and does not change the existing C-Clip.

Note: 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 38.)

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

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 38 for more information on server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

Read operations fail over by default in multicluster environments. Refer to the *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 133 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 133 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 an EMC 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 200 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 200 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 91 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 22.

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 94 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 38 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 *EMC Centera Programmer's Guide*.

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 22.

Description

This function sets a retention hold on a C-Clip, which effectively prevents the C-Clip from being deleted in EMC 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 38.)

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

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 91 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 38 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 *EMC Centera Programmer's Guide*.

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 73).
- `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 38.)

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 *EMC Centera Programmer's Guide*.

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 38.)

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 *EMC Centera Programmer's Guide*.

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 38.)

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 *EMC Centera Programmer's Guide*.

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 58, this function returns:

```
<C-CLIPID><REFID>
```

For example:
`42L0M726P04T2e7QU2445E81QBK7QU2445E81QBK42L0M726P04T2`.
Refer to the *EMC Centera Programmer's Guide* 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 *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 38 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

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 FPClip functions that retrieve information about a C-Clip:

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 *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 38 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

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 22.

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 *EMC Centera Programmer's Guide*.

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 22.

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 *EMC Centera Programmer's Guide*.

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 *EMC Centera Programmer's Guide*.

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 22.

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 22.

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 *EMC Centera Programmer's Guide*.

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 *EMC Centera Programmer's Guide*.

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 94 and "FPClip_SetRetentionClass" on page 91 for more information.

For more information on retention periods and classes, refer to the *EMC Centera Programmer's Guide*.

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 81 and "FPClip_EnableEBRWithPeriod" on page 83.

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

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 22.

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 22.

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 22.

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 22.

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 73 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 FPClip functions that manipulate a single tag from a C-Clip. Because these functions operate on the C-Clip level — and not on the tag level as described in "Tag functions" on page 136 — 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 37 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)

4

Tag Functions

This chapter describes the various types of FPTag API functions.

The main sections in this chapter are:

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 FPTag 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 22.

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 22.

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 FPTag functions that handle tag navigation within a C-Clip:

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

Figure 1 on page 148 shows how the tags are structured hierarchically and how the tag navigation functions operate if the C-Clip opens in tree mode.

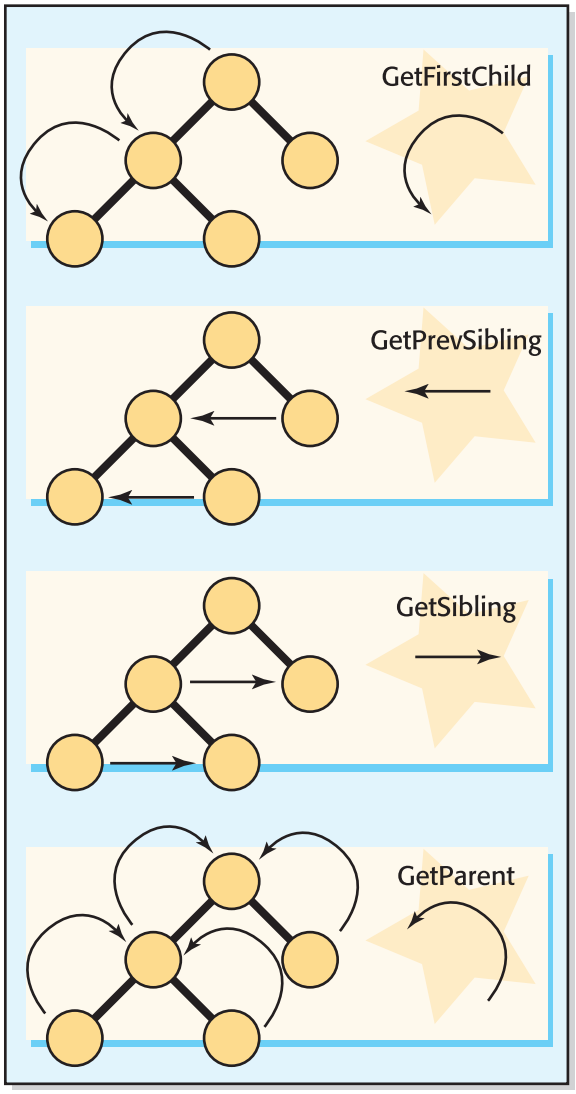

Figure 1 **Tag structure and navigation**

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 on page 148). 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 on page 148). 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 on page 148).

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 on page 148).

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 FPTag 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 22.

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 22.

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 22.

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 22.

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 22.

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 22.

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 22.

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 22.

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)

5 Blob Handling Functions

This chapter describes FPTag blob-handling functions.

The main section in this chapter is:

Blob handling functions

This section describes the FPTag 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 *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 200).

Note: Refer to "Stream creation functions" on page 203 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 38 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

Read operations fail over by default in multicluster environments. Refer to the *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 200). This function reads the data in chunks of 16 KB.

EMC recommends that you use `FPTag_BlobReadPartial` in conjunction with `FPStream_CreatePartialFileForInput` and `FPStream_CreatePartialFileForOutput`.

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 38 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

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.

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 *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 200).

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 61) 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 61) 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 *EMC Centera Programmer's Guide*.

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 38 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

Write operations do not fail over by default in multicluster environments. Refer to the *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

If collision avoidance is enabled at pool level, refer to "FPPool_SetClipID" on page 58, 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 an EMC 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 200.
- `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 EMC 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 EMC Centera's single-instance storage feature. For more information refer to the *EMC Centera Programmer's Guide* and "FPPool_SetClipID" on page 58. 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 `FP_OPTION_EMBED_DATA` has been set, 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 61).

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 200). 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 *EMC Centera Programmer's Guide*.

EMC recommends that you use `FPTag_BlobWritePartial` in conjunction with `FPStream_CreatePartialFileForInput` and `FPStream_CreatePartialFileForOutput`.

`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 38 for more information on the server capabilities. It is imperative that your application documentation contains server configuration details based on the *EMC Centera Online Help*.

Write operations do not fail over by default in multicluster environments. Refer to the *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

If collision avoidance is enabled at pool level, refer to "FPPool_SetClipID" on page 58, 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 an EMC 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 an EMC 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 200.
- `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 EMC 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 EMC Centera's single-instance storage feature. For more information refer to the *EMC Centera Programmer's Guide* and "FPPool_SetClipID" on page 58. 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 EMC 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)

6

Retention Class Functions

This chapter describes the FPRetentionClass API functions.

The main section in this chapter is:

Retention class functions

This section describes the FPRetentionClass 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 22.

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 22.

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)

7

Stream Functions

This chapter describes the FPStream API functions.

The main sections in this chapter are:

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 208.

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 SDK versions 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, to be filled by the SDK. 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-v1.2 SDK releases.

The following rules apply:

- Do not declare `prepareBufferProc`. If declared, set `mBuffer` to `NULL`.
- While the function `completeProc` can read the data from `mBuffer` to obtain the number of `mTransferLen` bytes, it is not allowed to change the 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` sets `mBuffer` to point to the buffer it manages. It also sets `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`. The number of `mTransferLen` bytes is 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 callback validation

In v3.2, the SDK performs generic stream verifications on certain fields that can be changed in application callbacks. To enable full validation, logging and the `FP_OPTION_STREAM_STRICT_MODE` option must be enabled. (This option is enabled by default.) If only logging is enabled, the SDK issues the appropriate warning message to the log. If the strict mode and logging are enabled and validation fails, the SDK generates either the `FP_STREAM_VALIDATION_ERR` or `FP_STREAM_BYTECOUNT_MISMATCH_ERR` error message, depending on the field.

The SDK validates the following `FPStreamInfo` fields:

- `mVersion`
- `MStreamLen`
- `mStreamPos`
- `mMarkerPos`
- `mReadFlag`

Table 19 on page 211 provides more information.

Note: You can disable the strict mode by setting the `FP_OPTION_STREAM_STRICT_MODE` environment variable to a non-zero value.

Stream creation functions

This section describes the FPStream functions for creating a stream:

- FPStream_CreateBufferForInput
- FPStream_CreateBufferForOutput
- FPStream_CreateFileForInput
- FPStream_CreateFileForOutput
- FPStream_CreateGenericStream
- FPStream_CreatePartialFileForInput
- FPStream_CreatePartialFileForOutput
- 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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)

- FP_STREAM_BYTECOUNT_MISMATCH_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 208 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)
- FP_STREAM_VALIDATION_ERR (program logic error)
- FP_STREAM_BYTECOUNT_MISMATCH_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 (Section "Stream handling functions" on page 231).

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. This value must be greater than 0.

  Note: EMC recommends a buffer size no larger than 1 MB, depending on the application requirements.

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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_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 231 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 exists, the function overwrites the content. 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.

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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

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 (`FPStreamInfo`), 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.

Note: If `FP_OPTION_STREAM_STRICT_MODE` is enabled (the default), the SDK validates these `FPStreamInfo` fields: `mVersion`, `MStreamLen`, `mStreamPos`, `mMarkerPos`, and `mReadFlag`. Table 19 on page 211 provides more information.

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 tables, **input stream** (to EMC Centera) refers to a stream from which the SDK has to read data, for example, when writing a blob to the cluster. **Output stream** (from EMC Centera) refers to a stream to which the SDK has to write data, for example, when reading a blob from the cluster.

Table 17 on page 209 and Table 18 on page 210 outline the input action or value required for the corresponding `FPStreamInfo` fields of each callback function. Table 19 on page 211 provides descriptions of these callback items.

Table 17 Generic stream to EMC Centera — FPStreamInfo/callbacks

FPStreamInfo ➤ Callbacks ▼	mVersion	mUserData	mStreamPos	mMarkerPos	mStreamLen	mAtEOF	mReadflag	mBuffer	mTransferLen
Initialization	Set by SDK	Initialize	0	0	Set to the total number of bytes that make up the data source if known, otherwise, use -1.	0	1	Null	0
PrepareBuffer Proc		As needed	+=mTransfer Length		Set to the total number of bytes that make up the data source if known, otherwise, use -1.	Set to true if this is the last buffer of data.		Allocate (if necessary) and populate transfer buffer.	Number of data bytes in buffer
CompleteProc		As needed						Free/release buffer (if applicable).	
SetMarkerProc		As needed							
ResetMarkerProc		As needed	=mMarkerPos	Use to reset current position of data source.					
CloseProc		As needed						Free/release buffer (if applicable).	

Table 18 Generic stream from EMC Centera — FPStreamInfo/callbacks

FPStreamInfo ➤ Callbacks ▼	mVersion	mUserData	mStreamPos	mMarkerPos	mStreamLen	mAtEOF	mReadflag	mBuffer	mTransferLen
Initialization	Set by SDK	Initialize	0	0	0	0	0	Null	0
PrepareBuffer Proc		As needed						Allocate buffer or set to 0 for SDK-supplied buffer.	Set max length of buffer.
CompleteProc		As needed	+=mTransfer Length			Check for true, signaling last data buffer.		Retrieve data from transfer buffer, free buffer if needed.	
SetMarkerProc									
ResetMarkerProc									
CloseProc		As needed						Free if using application buffer.	

Table 19 FPStreamInfo field descriptions per callback (1 of 12)

FPStreamInfo field	Description
PrepareBufferProc	
Callback behavior	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, 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. The SDK calls `myPrepareBuffer` prior to transferring data from the input stream to the EMC 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 EMC 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 to its length in `mTransferLen`. If the `inPrepareBufferProc` callback method returns a non-zero value, the SDK terminates the operation (for example, reading a blob) with an `FP_STREAM_VALIDATION_ERR` error and logs the appropriate information. No further processing can occur on that stream in the scope of that operation.
mVersion (short)	**For an input /output stream:** The version of the FPStreamInfo structure. Currently, the value of this field is **3**. Check this field for backward compatibility. Do not modify this field. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The callback has changed the mVersion field. It should never change in a callback.`

Table 19 FPStreamInfo field descriptions per callback (2 of 12)

FPStreamInfo field	Description
mUserData (void*)	Parameter passed unchanged from `FPStream_Create GenericStream` 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)	**For an input/output stream:** Initialized at 0. The current position in the stream where input occurs. Do not change this initial value before a read/write operation occurs. It can be updated by `myPrepareBuffer` if needed. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mStreamPos was not incremented by mTransferLen in the prepare callback.` In strict mode, the SDK checks to ensure the current stream position is incrementally based on the value specified in mStreamLen for the number of passes made through the stream. Validation of this field also occurs after a reset, which is equaled to 0, or after a resetMarker occurs if implemented by the application. In this case, the resetMarker is equaled to the value of mMarkerPos.
mMarkerPos (FPBool)	**For an input/output stream:** Initialized at 0. If the stream implementation uses an offset field to remember the marked position, `mMarkerPos` contains the position of the mark. `myResetMarker` sets the stream to that position. For file-based streams, `myResetMarker` does an `fseek()` on `mMarkerPos`. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mMarkerPos was changed incorrectly by the resetMarker callback.` In strict mode, the SDK checks for this update. An error results if the value of mMarkerPos does not match the current streamPos after the mark call was made.

Table 19 FPStreamInfo field descriptions per callback (3 of 12)

FPStreamInfo field	Description

mStreamLen ((FPLong)	**For an input stream:** The total length of the stream. This is –1 (unknown) by default and is updated by `myPrepareBuffer`. This field must be initialized and set to 0 before the first call to `prepareBuffer` is made (before `FPTag_BlobWrite` is called). **For an output stream:** Initialized at 0.
	
	This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning.
	
	`More data was written than the stream contains (known stream lengths only).`
	
	In strict mode, if the amount of data written to the stream does not match the amount of data in the stream, including any provision for an offset, the result is an `FP_STREAM_BYTECOUNT_MISMATCH_ERR` error and the following message:
	
	`Overwrite detected. More data was written than the stream contains (known stream lengths only).`

mAtEOF (FPBool)	**For an input/output stream:** Initialized at 0. Indicates whether or not `myStream` has reached the end of the stream. `myPrepareBuffer` returns true in this field if the last segment is read.

Table 19 FPStreamInfo field descriptions per callback (4 of 12)

FPStreamInfo field	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. **For an input stream:** Initialized to true (1). **For an output stream:** Initialized to false (0). This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The callback has changed the mReadFlag. It should never change in a callback.` In strict mode, the SDK detects that a change to the stream type was made after the operation initialized.
mBuffer (void*)	**For an input stream:** Initialized at NULL. `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. **For an output stream:** Initialized at NULL. This field should either be set to a stream-managed buffer or left set to NULL.
mTransferLen (FPLong)	**For an input stream:** Initialized at 0. `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, mTransferLen = 0. If mTransferLen = 0 and `mAtEOF` is true, the end of the data stream has been reached. **For an output stream:** Initialized at 0. If `myPrepareBuffer` manages the output buffer, `mTransferLen` is 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, this field is unused.

Table 19 FPStreamInfo field descriptions per callback (5 of 12)

FPStreamInfo field	Description
CompleteProc	
Callback behavior	The generic stream calls this method when the buffer 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`, 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. **For an output stream:** `myComplete` actually transfers the data pointed to by `mBuffer` to the output device. For a file, this might translate into an `fwrite()` operation. The `mBuffer` pointer can either be provided by the stream or by the SDK. The end of the stream behaves as follows: • If the application allocated its buffer in `prepareBufferProc`, the last `completeProc` callback has `mTransferLen` >= 0 and `mAtEOF` is true. • If the application did not allocate its buffer, the last call to `completeProc` always has `mTransferLen` = 0 and `mAtEOF` is true. **Note:** If the SDK provides the `mBuffer` pointer to the stream to read the output data, the callback function should never change this data. **For 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. **Note:** If the `inCompleteProc` callback method returns a non-zero value, the SDK terminates the operation (for example, reading a blob) with an `FP_STREAM_VALIDATION_ERR` error and logs the appropriate information. No further processing can occur on that stream in the scope of that operation.
mVersion (short)	See "mVersion" on page 211.
mUserData (void*)	See "mUserData" on page 212.

Table 19 FPStreamInfo field descriptions per callback (6 of 12)

FPStreamInfo field	Description
mStreamPos (FPLong)	**For an input/output stream:** Initialized at 0. The current position in the stream where input or output occurs. This field is updated by `myComplete` if needed. It initializes when `myStream` is created. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mStreamPos was not incremented by mTransferLen in the complete callback.`
mMarkerPos (FPBool)	**For an input/output stream:** Initialized at 0. If the stream implementation uses an offset field to remember the marked position, `mMarkerPos` contains the position of the mark. `myResetMarker` sets the stream to that position. For file-based streams, `myResetMarker` does an `fseek()` on `mMarkerPos`. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mMarkerPos was changed incorrectly by the resetMarker callback.` In strict mode, the SDK checks for this update. An error results if the value of mMarkerPos does not match the current streamPos after the mark call was made.

Table 19 FPStreamInfo field descriptions per callback (7 of 12)

FPStreamInfo field	Description

mStreamLen ((FPLong)	The total length of the stream.
	
	For an input stream:
	Initialized at –1 by default (unknown) and is updated by `myComplete`. If known, set to the total number of bytes of the data source. It must be initialized and set to 0 before the first call to `prepareBuffer` is made (before `FPTag_BlobWrite` is called).
	For an output stream:
	Initialized at 0.
	
	This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning.
	
	`More data was written than the stream contains (known stream lengths only).`
	
	In strict mode, if the amount of data written to the stream does not match the amount of data in the stream, including any provision for an offset, the result is an `FP_STREAM_BYTECOUNT_MISMATCH_ERR` error and the following message:
	
	`Overwrite detected. More data was written than the stream contains (known stream lengths only).`

mAtEOF (FPBool)	Indicates that the last buffer of data will now be written. If `mTransferLen` = 0, no data is passed. **For an input stream:** Initialized at 0. **For an output stream:** Initialized at 0.
mReadflag (FPBool)	See "mReadflag" on page 214.
mBuffer (void*)	**For an input stream:** Initialized to NULL. Contains a pointer to the data that has been read (as provided by an earlier call from `myPrepareBuffer`). **For an output stream:** Initialized to NULL. 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.

Table 19 FPStreamInfo field descriptions per callback (8 of 12)

FPStreamInfo field	Description
mTransferLen (FPLong)	**For an input stream:** Initialized at 0. Contains the number of bytes that have been read (as provided by an earlier call from `myPrepareBuffer`). **For an output stream:** Initialized at 0. Contains the number of bytes that `mBuffer` points to. This value can be 0 (in which case `mAtEOF` is true).
SetMarkerProc	
Callback behavior	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`, 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. 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, 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.
mUserData (void*)	See "mUserData" on page 212.

Table 19 FPStreamInfo field descriptions per callback (9 of 12)

FPStreamInfo field	Description
mStreamPos (FPLong)	**For an input/output stream:** Initialized at 0. The current position in the stream where input or output occurs. `mySetMarker` updates this field to reflect the current position in the stream. For a file-based stream, this is often an `ftell()` call. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mStreamPos has been incorrectly changed by the setMarker callback.`
mReadflag (FPBool)	See "mReadflag" on page 214.
mMarkerPos (FPLong)	**For an input/output stream:** Initialized at 0. If the stream implementation uses an offset field to remember the marked position, `mMarkerPos` contains the position of the mark. `myResetMarker` sets the stream to that position. For file-based streams, `myResetMarker` does an `fseek()` on `mMarkerPos`. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mMarkerPos does not equal the mStreamPos after a setMarker callback.` In strict mode, the SDK checks for this update. An error results if the value of mMarkerPos does not match the current streamPos after the mark call was made.

Table 19 FPStreamInfo field descriptions per callback (10 of 12)

FPStreamInfo field	Description
ResetMarkerProc	
Callback behavior	This method tells the stream to go back to the marked position in the stream (`mMarkerPos`). If you name your function `myResetMarker`, 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. 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. If the SDK attempts to return to a previously marked position in the stream and the stream returns `FP_OPERATION_REQUIRES_MARK` when the attempt is made, the SDK cannot continue with the operation and propagates the `FP_OPERATION_REQUIRES_MARK` error to the application.
mVersion (short)	See "mVersion" on page 211.
mUserData (void*)	See "mUserData" on page 212.
mStreamPos (FPLong)	**For an input/output stream:** Initialized at 0. The current position in the stream where input or output occurs. `mySetMarker` updates this field to reflect the current position in the stream. For a file-based stream, this is often an `ftell()` call. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mStreamPos does not equal the mStreamPos after a resetMarker callback.`

Table 19 FPStreamInfo field descriptions per callback (11 of 12)

FPStreamInfo field	Description
mMarkerPos (FPBool)	**For an input/output stream:** Initialized at 0. If the stream implementation uses an offset field to remember the marked position, `mMarkerPos` contains the position of the mark. `myResetMarker` sets the stream to that position. For file-based streams, `myResetMarker` does an `fseek()` on `mMarkerPos`. This is an SDK-validated field. If logging and the `FP_OPTION_STREAM_STRICT_MODE` option (enabled by default) are both turned on and the validation fails, the `FP_STREAM_VALIDATION_ERR` is thrown with the following error message. If only strict mode is on (logging is off) and an error occurs, only the error message is thrown but with no logging. If an error occurs with just logging enabled, the SDK sends the same message to the log, but only as a warning. `The mMarkerPos was changed incorrectly by the resetMarker callback.` In strict mode, the SDK checks for this update. An error results if the value of mMarkerPos does not match the current streamPos after the mark call was made.
mReadflag (FPBool)	See "mReadflag" on page 214.
CloseProc	
Callback behavior	**For an input/output stream:** Initialized at 0. 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`, 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. 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, `NULL` can be passed to `FPStream_CreateGenericStream`.

Table 19 FPStreamInfo field descriptions per callback (12 of 12)

FPStreamInfo field	Description
mVersion (short)	See "mVersion" on page 211.
mUserData (void*)	See "mUserData" on page 212.
mStreamLen (FPLong)	**For an input stream only** If `mReadFlag` = true (1) , then mStreamLen applies. See page 213.

Example

The following 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 buffer size. 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;
```

1. Or any other error value. The error value will be returned to the application.

```
  return ENOERR;
}
```

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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

FPStream_CreatePartialFileForInput

Syntax

```
FPStream_CreatePartialFileForInput (const char
*inFilePath, const char *inPerm, const long inBuffSize,
const long inOffset, const long inLength)
```

Return value

`FPStreamRef`

Input parameters

```
const char *inFilePath, const char *inPerm, const long
inBuffSize, const long inOffset, const long inLength
```

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 Section "Stream handling functions" on page 231 for more information).

This API call allows applications to simplify the handling of large files for input into EMC Centera by allowing a large file to be divided into multiple streams and processed with multiple threads via `FPTag_BlobWritePartial()`.

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. This size represents the amount of data to retrieve or cache from the file into memory each time the file is touched. This value must be greater than 0.

  Note: EMC recommends a buffer size no larger than 1 MB, depending on the application requirements.

- `const long inOffset`
  `inOffset` is the offset into the file (in bytes) from which EMC Centera starts the read.

- `const long inLength`
  `inLength` is the amount of data (in bytes) from the offset to be transferred in the stream. Use `FP_STREAM_EOF` to request all remaining data in the file from the given offset.

Example

This example shows how to stream the data to a pool.

```
  { FPStreamRef vStream = FPStream_CreatePartialFileForInput (pPath, "rb",
16*1024, 0, 100);
// Write 100 bytes from offset 0
// open a new stream
    if (vStream != 0)
      { FPTag_BlobWritePartial (vFileTag, vStream, FP_OPTION_CLIENT_CALCID,
seqID);
// 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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

FPStream_CreatePartialFileForOutput

Syntax

```
FPStream_CreatePartialFileForOutput (const char
*inFilePath, const char *inPerm, const long inBuffSize,
const long inOffset, const long inLength, const long
inMaxFileLength)
```

Return value

`FPStreamRef`

Input parameters

`const char *inFilePath, const char *inPerm, const long inBuffSize, const long inOffset, const long inLength`

Concurrency requirement

This function is thread safe, however, with an exception. This call is not thread safe when it writes to the same file from multiple threads. In this case, if multiple streams are created and open the same file for writing, an `FP_FILESYS_ERR` results for all created streams except the first stream (until the initial stream closes). Only one stream may write to a given file at a time.

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 most of the stream handling functions (refer to Section "Stream handling functions" on page 231 for more information).

This API call allows applications to simplify the handling of large files for output from EMC Centera by allowing a large file to be divided into multiple streams and processed with multiple threads via `FPTag_BlobWritePartial()`.

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. For partial writes, **ab+** is typically used. However, you can use any 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 exists, the function overwrites the content. 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.

- `const long inBuffSize`
  `inBuffSize` is the size of the buffer (in bytes) that is used when reading the file. This size represents the amount of data to retrieve or cache from the data source into memory. This value must be greater than 0.

  Note: EMC recommends a buffer size no larger than 1 MB, depending on the application requirements.

- `const long inOffset`
  `inOffset` is the offset into the file (in bytes) in which EMC Centera begins the write.
- `const long inLength`
  `inLength` is the maximum amount of data (in bytes) this stream can accept. A value of -1 indicates that this stream can accept data up to the value specified in `inMaxFileLength`. If the `inMaxFileLength` is out of bounds (for example, `!=-1 AND <-1 or >= (inFileLength-inOffset)`), the SDK generates an `FP_PARAM_ERR` and returns a `NULL FPStreamRef`.
- `const long inMaxFileLength`
  `inMaxFileLength` is the maximum overall amount of data that can be written to the underlying file, which the stream accesses. This defines the maximum size a file may be allowed to grow. If the file size on the disk exceeds this size, the file is truncated. A value of -1 indicates that there is no overall maximum data length.

Example This example shows how to stream the data to a pool.

```
  { FPStreamRef vStream = FPStream_CreatePartialFileForOutput (pPath, "ab+",
16*1024, 0, 100, vFileMaxSize);
// Read 100 bytes to offset 0
// open a new stream
    if (vStream != 0)
      { FPTag_BlobReadPartial (vFileTag, vStream, 0, 100,
FP_OPTION_DEFAULT_OPTIONS);
```

```
// 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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_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 FPStream functions for handling 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 *EMC Centera Programmer's Guide*, 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_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

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 208 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 *EMC Centera Programmer's Guide*, 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_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

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 *EMC Centera Programmer's Guide*, 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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

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 *EMC Centera Programmer's Guide*, 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)
- `FP_STREAM_VALIDATION_ERR` (program logic error)
- `FP_STREAM_BYTECOUNT_MISMATCH_ERR` (program logic error)

8

Query Functions

This chapter describes the various types of FPQuery API functions.

The main sections in this chapter are:

Query functions

The Access API provides functions that query C-Clips — both existing and deleted (*reflections*) — stored on an EMC Centera cluster.

The query feature is intended for backup applications and not as a general-purpose application feature. Refer to the *EMC Centera Programmer's Guide* for more information on the query feature.

This section describes the following types of query functions:

- Query expression functions
- Pool query functions
- Query result functions

Query expression functions

This section describes the query expression functions that are used to create, define, close, and manipulate (elements of) a query expression:

- FPQueryExpression_Close
- FPQueryExpression_Create
- FPQueryExpression_DeselectField
- FPQueryExpression_GetEndTime
- FPQueryExpression_GetStartTime
- FPQueryExpression_GetType
- FPQueryExpression_IsFieldSelected
- FPQueryExpression_SelectField
- FPQueryExpression_SetEndTime
- FPQueryExpression_SetStartTime
- FPQueryExpression_SetType

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 = FPQueryExpression_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 246 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 246 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 130. Table 20 on page 246 lists the standard C-Clip attributes. Table 21 on page 247 lists the standard reflection attributes.

Table 20 Standard C-Clip description attributes (1 of 2)

Attribute name	Description
name	The name of the C-Clip. Refer to "FPClip_AuditedDelete" on page 73 and "FPClip_GetName" on page 113.
creation.date	The timestamp when the C-Clip was created. Refer to "FPClip_AuditedDelete" on page 73 and "FPClip_GetCreationDate" on page 108.
modification.date	The timestamp when the C-Clip was written to the cluster. Refer to "FPClip_Write" on page 101.
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 20 Standard C-Clip description attributes (2 of 2)

Attribute name	Description
numfiles	The number of referenced blobs. Refer to "FPClip_GetNumBlobs" on page 114.
totalsize	The total size of referenced blobs. This size does not include the CDF itself. Refer to "FPClip_GetRetentionClassName" on page 117.
prev.clip	The Clip ID of the C-Clip that was modified to create this C-ClipI
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 115.
sdk.version	The version of the SDK used to create the C-Clip. Refer to "FPPool_GetComponentVersion" on page 45.
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 94.
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 91.

Table 21 Standard reflection description attributes (1 of 2)

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 21 Standard reflection description attributes (2 of 2)

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 73 for more information.
All standard attributes of the deleted C-Clip.	A reflection retains all the standard 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 EMC Centera cluster that stores the C-Clips. A value of `-1` or `FP_QUERY_END_TIME_UNBOUNDED` corresponds to the current time.

Note: For more information on timestamps, refer to Chapter 6, *Best Practices,* in the *EMC Centera Programmer's Guide*.

Example

```
FPQueryExpression_SetEndTime (myQueryExp,
FP_QUERY_END_TIME_UNBOUNDED);
```

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 EMC Centera cluster that stores the C-Clips. If `inTime` is `0` or `FP_QUERY_START_TIME_UNBOUNDED`, 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 *EMC Centera Programmer's Guide*.

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)

Pool query functions

This section describes the pool query functions that act upon an existing pool query:

- FPPoolQuery_Close
- FPPoolQuery_FetchResult
- FPPoolQuery_GetPoolRef
- FPPoolQuery_Open

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 261.

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 253 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 240 for more information.

Queries do not fail over by default in multicluster environments. Refer to the *EMC Centera Programmer's Guide* for more information on multicluster failover. To configure failover behavior, refer to "FPPool_SetGlobalOption" on page 61.

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 EMC 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

Query result functions

This section describes the query result functions that retrieve a query result, including one that closes a query result.

- FPQueryResult_Close
- FPQueryResult_GetClipID
- FPQueryResult_GetField
- FPQueryResult_GetResultCode
- FPQueryResult_GetTimestamp
- FPQueryResult_GetType

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 EMC 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)

9 Monitoring Functions

This chapter describes the monitoring (MoPI) API functions.

The main section in this chapter is:

Monitoring functions

The Monitor API (MoPI) lets you gather monitoring information from the EMC 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 EMC 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 EMC 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 EMC 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 EMC 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 EMC 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 22.

Description

This function creates a new object to monitor the state of the EMC 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)

10

Time Functions

This chapter describes the time API functions.

The main section in this chapter is:

Time functions

This section describes the following time formats used by the SDK to convert EMC 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 22.

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 22.

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 22.

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 22.

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)

11

Logging Functions

This chapter describes the logging API functions, including logging behavior and the use of logging environment variables.

The main sections in this chapter are:

Logging overview

For application debugging of supported platforms, the SDK provides thread-safe logging of its activities via log API functions or logging environment variables. Logging does not create new threads.

The SDK logging interface allows logging to be configurable via API, configuration file (which may be optionally polled), and environment variables. Application developers can dynamically control logging features with a control class. This logging interface consists of the `FPLogging` mechanism and the `FPLogState` object.

FPLogging mechanism

`FPLogging` is a logging framework that enables SDK logging and logging control on each cluster. The `FPLogging` mechanism uses the `FPLogState` object, which is the control class that defines the conditions for determining logging behavior. `FPLogging` is also the factory from which other instances of `FPLogState` objects may be created.

You can use various log API functions to perform the following actions:

- Start and stop logging anytime during runtime
- Modify SDK log settings in (dynamic) real time
- Write application-defined log messages to the SDK log file
- Use a callback function to capture SDK log data directly for application logs

When checking for log state settings, the `FPLogging` mechanism observes the following order of priority, from highest to lowest:

1. `FPLogState` settings applied by `FPLogging_Start()` calls
2. `FPLogState.cfg` in the working directory
3. `FP_LOG_STATE_PATH` environment variable
4. Logging environment variables

For example, during application startup, `FPLogging` first checks for the presence of the `FPLogState.cfg` properties file in the working directory. If this file exists, the SDK automatically reads and uses the settings contained in that file, including the log path for the logging

output. No attempt is made to read any logging environment variables unless `FPLogState.cfg` is absent from the working directory.

If this is the case and environment variables are to be used, the log state settings of the configuration file defined in `FP_LOG_STATE_PATH` (if specified) take precedence. That is, the SDK ignores the log path specified in the `FP_LOGPATH` environment variable.

If `FP_LOG_STATE_PATH` is not defined, the SDK uses the values specified in the logging environment variables. For example, the path set in the `FP_LOGPATH` is where the SDK directs the logging output. "Polling behavior" on page 288 describes how the SDK handles poll events.

FPLogState

`FPLogState` is the control class object that defines the conditions for logging behavior and logging control. You create an `FPLogState` object by calling `FPLogging_CreateLogState()`, which is then passed to the `FPLogging` mechanism.

This log state object contains the following logging configurations:

- `FP_LOGPATH`
- `FP_LOGKEEP`
- `FP_LOGLEVEL`
- `FP_LOGFILTER`
- `FP_LOGFORMAT`
- `FP_LOG_DISABLE_CALLBACK`
- `FP_LOG_MAX_OVERFLOWS`
- `FP_LOG_MAX_SIZE`
- `FP_LOG_STATE_POLL_INTERVAL`

Note: "Environment variables" on page 313 provides definitions of each configuration setting.

After creating an `FPLogState` object, you can save it by calling `FPLogState_Save()`. You can set the path to this properties file in the `FP_LOG_STATE_PATH` environment variable. "Example: FPLogState configuration file" on page 317 shows an example.

Note: If you explicitly load an existing log state, the settings associated with that log state override any settings found in the environment variables (as described in "FPLogging_OpenLogState" on page 293).

Logging environment variables

The set of logging environment variables (as listed in "Environment variables" on page 313) allows application administrators to control logging behavior without having to change or write (additional) application code. The application administrator can specify the following:

- Dynamically modify log settings (via configuration file) for an application without requiring an application restart.
- Control the interval at which the SDK polls and applies the log configuration file at runtime.
- Limit the size at which log files are allowed to grow.
- Retain a fixed number of backup log files.

"Environment variables" on page 313 provides definitions of each configuration setting.

Note: For optimal debugging practice, EMC recommends the use of the logging API calls to manage the SDK logging.

Polling behavior

The polling of a log state file automatically occurs regardless if logging is turned on or off. The conditions that affect logging polling behavior are as follows:

- If you specify a log state configuration file with either the `FPLogging_OpenLogState()` API call or `FP_LOG_STATE_PATH` environment variable, the SDK polls the file at the interval defined in the configuration file (or the 5-minute default if unspecified) or in the `FP_LOG_STATE_POLL_INTERVAL` environment variable. If the log interval is set to 0, the SDK polls the log configuration file prior to every log event.
- If you modify the `FP_LOGPATH` field of an active configuration file during a poll event, the state dynamically changes in real time. For example, you change the log file path by editing it in the configuration file and saving it.
- Similarly, if you modify the path to a filename in the `FP_LOGPATH` field during a poll event, the SDK immediately redirects the log output to the new file based on the settings specified in `FP_LOGFORMAT` and `FP_LOGKEEP`.

- When a poll event interval occurs, any updates made to the following configuration file fields are applied:
  - FP_LOGPATH
  - FP_LOGLEVEL
  - FP_LOGFORMAT
  - FP_LOGFILTER
  - FP_LOGKEEP
  - FP_LOG_DISABLE_CALLBACK
  - FP_LOG_STATE_POLL_INTERVAL
  - FP_LOG_MAX_SIZE
  - FP_LOG_MAX_OVERFLOWS

Log file format

The log file contains the following information:

- MS 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.
- FormattedTime — The current time of the logged message in HH:MM:SS.ms format.
- LogLevel — The verbosity level of the message to indicate how much information to include in the log (for example, `log` or `debug`).
- PID.Thread ID — The process ID of the message and the ID of the thread that logged the message.
- Component/Module — 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.
- Message — The actual message string.

Application strings in SDK log

Using `FPLogging_Log()`, applications can directly insert application-defined strings into the SDK log. You can filter these messages at the application level by including the `APP` component in the `FPLogState_SetLogFilter()` API call or alternatively, by setting it in the `FP_LOGFILTER` environment variable. "FPLogging_Log" on page 292 and "Environment variables" on page 313 provide more information.

Redirecting log output

Conversely, applications can integrate SDK logging into their own log by redirecting the output of the SDK logging activity. To do this, applications use the `FPLogging_RegisterCallback()` call (page 294) to establish a callback routine that allows the SDK to pass the logging string to the registered callback. It can be disabled via the `FPLogState_SetDisableCallback()` function (page 305).

The `FP_LOG_DISABLE_CALLBACK` environment variable (page 313) can be set to disable or enable the callback.

FPLogging functions

The FPLogging functions determine the initial state of the `FPLogState` object and control logging operations.

- `FPLogging_CreateLogState`
- `FPLogging_Log`
- `FPLogging_OpenLogState`
- `FPLogging_RegisterCallback`
- `FPLogging_Start`
- `FPLogging_Stop`
- `FPLogState_Delete`
- `FPLogState_Save`

FPLogging_CreateLogState

Syntax `FPLogging_CreateLogState()`

Return value `FPLogStateRef`

Concurrency requirement This function is thread safe.

Description This function creates an `FPLogState` object. This object contains all default field values. When building a new `FPLogState` object, the SDK checks to see if any logging environment variables are set. If yes, it uses the values of the set environment variables to initialize the corresponding fields in the `FPLogState` object. If no logging environment variables are set, the object is given the default field values.

When you no longer need the reference, use `FPLogState_Delete()` to release the resources allocated by this method.

Example

```
// Create a log state object (all defaults)
FPLogStateRef vLogState = FPLogging_CreateLogState();
```

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_OUT_OF_MEMORY_ERR` (program logic error)

FPLogging_Log

Syntax

```
FPLogging_Log (const FPLogLevel inLogLevel, const char
*inMessage)
```

Return value

```
void
```

Input parameters

```
const FPLogLevel inLogLevel, const char *inMessage
```

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 routines" on page 24.

Description

This function writes a message to the configured SDK log components at the specified logging level.

Parameters

- `const FPLogLevel inLogLevel`
  The level at which to log the message. The following log levels are available:
  - `FP_LOGLEVEL_ERROR`: Log error messages
  - `FP_LOGLEVEL_WARN`: Log warnings and error messages.
  - `FP_LOGLEVEL_LOG`: Log API calls, warnings, and errors
  - `FP_LOGLEVEL_DEBUG`: Debug log level
- `const char *inMessage`
  A null-terminated string to write to the log.

Example

```
// Write a message to the FP_LOGGING_COMPONENT_APP
component in the log
FPLogging_Log(FP_LOGLEVEL_ERROR, "This is a fatal
error");
```

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)

FPLogging_OpenLogState

Syntax `FPLogging_OpenLogState(char * inPathName)`

Return value `FPLogStateRef`

Input parameters `char * inPathName`

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 22.

Description This function loads an `FPLogState` properties file into an `FPLogState` object. When you no longer need the reference, use `FPLogState_Delete` to release the resources allocated by this method.

Note: If you use this method to open an `FPLogState` file from disk, the log state file is polled every 5 minutes (default). You can specify (or disable) the poll interval using the `FPLogState_SetPollInterval` API call.

Parameters

- `char * inPathName`
  The full path to a serialized `FPLogState` properties file.

Example

```
// Open an existing log state object from an FPLogState
properties file
FPLogStateRef vLogState =
FPLogging_OpenLogState("C:\\MyLogState.cfg");
```

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)
- `FP_OUT_OF_MEMORY_ERR` (client error)

FPLogging_RegisterCallback

Syntax

```
FPLogging_RegisterCallback(FPLogProc inCallback)
```

Return value

```
void
```

Input parameters

```
FPLogProc inCallback
```

Concurrency requirement

This function is thread safe.

Description

This function registers an application callback method with the logging mechanism.

If you specify this callback, the logging mechanism uses the method pointed to by `inCallback` to log an event, instead of directing the output to a file. If you already enabled logging before calling this method, all log output is redirected to the callback and does not appear in the log file. If you disable the callback, subsequent log output continues to appear in the log file.

You can disable the callback by passing `NULL` to this method, or by using the `disable_callback` field in an active `FPLogState` object (see "FPLogState_SetDisableCallback" on page 305).

Parameters

- `FPLogProc inCallback`
  A reference to the logging callback method to log an event.

Example

```
// Define a logging callback method
FPInt appStdoutLogger(char* inString)
{
    printf(inString);
    return(0);
}

// Register the appStdoutLogger callback with the logging
mechanism
// The appStdoutLogger callback will be called instead of
being logged to the file.
FPLogging_RegisterCallback(appStdoutLogger);
```

Error handling

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

- `FP_LOGGING_CALLBACK_ERR`

FPLogging_Start

Syntax

```
FPLogging_Start(FPLogStateRef inLogState)
```

Return value

```
void
```

Input parameters

```
FPLogStateRef inLogState
```

Concurrency requirement

This function is thread safe.

Description

This function enables the logging mechanism based on the given log configuration. If logging is already configured via environment variables, this method restarts the logging mechanism using the new settings (potentially closing the old log file and opening a new one).

Parameters

- `FPLogStateRef inLogState`
  A reference to an `FPLogState` object containing log configuration details. Pass `FPREF_NULL` to use all default state settings.

Example

```
// Start logging using the given FPLogState settings
FPLogging_Start(vLogState);
```

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)

FPLogging_Stop

Syntax

```
FPLogging_Stop()
```

Return value

```
void
```

Concurrency requirement

This function is thread safe.

Description

This function disables the logging mechanism. If any log files created with logging are open, this call flushes and closes them. Similarly, any polling by log state files is also disabled.

Example

```
// Stop all logging
FPLogging_Stop();
```

FPLogState_Delete

Syntax

```
FPLogState_Delete(FPLogStateRef inLogState)
```

Return value

```
void
```

Input parameters

```
FPLogStateRef inLogState)
```

Concurrency requirement

This function is thread safe.

Description

This function releases all resources associated with an `FPLogState` object. Use this method in combination with `FPLogging_CreateLogState()` and `FPLogging_OpenLogState()` to delete an allocated `FPLogState` object.

Parameters

- `FPLogStateRef inLogState`
  A reference to an `FPLogState` object.

Example

```
// Delete the FPLogState object when done
FPLogState_Delete(vLogState);
```

FPLogState_Save

Syntax

```
FPLogState_Save(FPLogStateRef inLogState, char
*inPathName)
```

Return value

```
void
```

Input parameters

```
FPLogStateRef inLogState
```

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 22.

Description

This function saves the settings of an FPLogState object to a properties (configuration) file.

Parameters

- `FPLogStateRef inLogState`
  A reference to an `FPLogState` object.
- `char * inPathName`
  The full path to a serialized `FPLogState` properties file.

Example

```
// Save an FPLogState properties file to disk based on
these settings
FPLogState_Save(vLogState, "C:\\MyLogState.cfg");
```

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_FILESYS_ERR` (program logic error)

FPLogState functions

The FPLogState Get functions return a specific logging value:

- FPLogState_GetAppendMode
- FPLogState_GetDisableCallback
- FPLogState_GetLogFilter
- FPLogState_GetLogFormat
- FPLogState_GetLogLevel
- FPLogState_GetLogPath
- FPLogState_GetMaxLogSize
- FPLogState_GetMaxOverflows
- FPLogState_GetPollInterval

The FPLogState Set functions update the `FPLogState` control class object with the latest logging settings:

- FPLogState_SetAppendMode
- FPLogState_SetDisableCallback
- FPLogState_SetLogFilter
- FPLogState_SetLogFormat
- FPLogState_SetLogLevel
- FPLogState_SetLogPath
- FPLogState_SetMaxLogSize
- FPLogState_SetMaxOverflows
- FPLogState_SetPollInterval

FPLogState_GetAppendMode

Syntax `FPLogState_GetAppendMode(FPLogStateRef inLogState)`

Return value `FPBool`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the append mode field of the `FPLogState` object. See "FPLogState_SetAppendMode" on page 304.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPBool vAppend = FPLogState_GetAppendMode (vLogState);
```

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)

FPLogState_GetDisableCallback

Syntax `FPLogState_GetDisableCallback(FPLogStateRef inLogState)`

Return value `FPBool`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the disable callback field of the `FPLogState` object. See "FPLogState_SetDisableCallback" on page 305.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPBool vDisableCallback = FPLogState_GetDisableCallback
(vLogState);
```

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)

FPLogState_GetLogFilter

Syntax `FPLogState_GetLogFilter(FPLogStateRef inLogState)`

Return value `FPInt`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the filter field of the `FPLogState` object. See "FPLogState_SetLogFilter" on page 306.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPInt vFilterMask = FPLogState_GetLogFilter (vLogState);
```

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)

FPLogState_GetLogFormat

Syntax `FPLogState_GetLogFormat(FPLogStateRef inLogState)`

Return value `FPLogFormat`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the formatter field of the `FPLogState` object. See "FPLogState_SetLogFormat" on page 307.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPLogFormat vFormat = FPLogState_GetLogFormat
(vLogState);
```

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)

FPLogState_GetLogLevel

Syntax
```
FPLogState_GetLogLevel(FPLogStateRef inLogState)
```

Return value `FPLogLevel`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the log level field of the `FPLogState` object. See "FPLogState_SetLogLevel" on page 308.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example
```
FPLogLevel vLevel = FPLogState_GetLogLevel (vLogState);
```

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)

FPLogState_GetLogPath

Syntax
```
FPLogState_GetLogPath(FPLogStateRef ioLogState)
```

Return value
```
const char*
```

Input parameters `FPLogStateRef ioLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the log path field of the `FPLogState` object. See "FPLogState_SetLogPath" on page 309.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
const char* vLogPath = FPLogState_GetLogPath(vLogState);
```

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)

FPLogState_GetMaxLogSize

Syntax

```
FPLogState_GetMaxLogSize(FPLogStateRef inLogState)
```

Return value

`FPLong`

Input parameters

`FPLogStateRef inLogState`

Concurrency requirement

This function is thread safe.

Description

This function gets the current value of the max log size field of the `FPLogState` object. See "FPLogState_SetMaxLogSize" on page 310.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPLong vMaxLogSize = FPLogState_GetMaxLogSize
(vLogState);
```

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)

FPLogState_GetMaxOverflows

Syntax `FPLogState_GetMaxOverflows(FPLogStateRef inLogState)`

Return value `FPInt`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the max overflows field of the `FPLogState` object. See "FPLogState_SetMaxOverflows" on page 311.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPInt vMaxOverflows = FPLogState_GetMaxOverflows
(vLogState);
```

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)

FPLogState_GetPollInterval

Syntax `FPLogState_GetPollInterval(FPLogStateRef inLogState)`

Return value `FPInt`

Input parameters `FPLogStateRef inLogState`

Concurrency requirement This function is thread safe.

Description This function gets the current value of the poll interval field of the `FPLogState` object. See "FPLogState_SetPollInterval" on page 312.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.

Example

```
FPInt vPollIntervalInMin = FPLogState_GetPollInterval
(vLogState);
```

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)

FPLogState_SetAppendMode

Syntax

```
FPLogState_SetAppendMode(FPLogStateRef ioLogState,
FPBool inAppendMode)
```

Return value `void`

Input parameters `FPLogStateRef ioLogState, FPBool inAppendMode`

Concurrency requirement This function is thread safe.

Description This function updates the `AppendMode` field of the `FPLogState` object. This field indicates whether or not the logging mechanism is to append any existing file found at the given `LogPath`.

You can initialize the default value of `LogPath` by setting the `FP_LOGKEEP` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPBool inAppendMode`
  If set to true, an append is allowed, otherwise, false for an overwrite (default).

Example

```
FPLogState_SetAppendMode(vLogState, false); // overwrite
```

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)

FPLogState_SetDisableCallback

Syntax

```
FPLogState_SetDisableCallback(FPLogStateRef ioLogState,
FPBool inDisableCallback)
```

Return value

`void`

Input parameters

`FPLogStateRef ioLogState, FPBool inDisableCallback`

Concurrency requirement

This function is thread safe.

Description

This function updates the `disable callback` field of the FPLogState object, and disables (or allows) the use of any application callback registered with the `FPLogging_RegisterCallback` API method.

You can initialize the default value of `disable callback` by setting the `FP_LOG_DISABLE_CALLBACK` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPBool inDisableCallback`
  If set to true, registered callback is disabled. If set to false, registered callback is enabled (default).

Example

```
FPLogState_SetDisableCallback(vLogState, false);
// Allow callback registration
```

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)

FPLogState_SetLogFilter

Syntax

```
FPLogState_SetLogFilter(FPLogStateRef ioLogState, FPInt
inLogComponents)
```

Return value

```
void
```

Input parameters

```
FPLogStateRef ioLogState, FPInt inLogComponents
```

Concurrency requirement

This function is thread safe.

Description

This function updates the `LogFilter` field of the `FPLogState` object, and defines which SDK components are to be logged.

You can initialize the default value of `LogFilter` by setting the `FP_LOGFILTER` environment variable. For the list of possible SDK components to include for logging, see `FP_LOGFILTER` inTable 22 on page 313.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPInt inLogComponents`
  A bitmask that defines the active components. To enable full logging, use the `FP_LOGGING_COMPONENT_ALL` mask. To log only API level calls, use the `FP_LOGGING_COMPONENT_API` mask.

Example

```
FPLogState_SetLogFilter(vLogState,
FP_LOGGING_COMPONENT_ALL);
```

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)

FPLogState_SetLogFormat

Syntax

```
FPLogState_SetLogFormat(FPLogStateRef ioLogState,
FPLogFormat inLogFormat)
```

Return value

`void`

Input parameters

`FPLogStateRef ioLogState, FPLogFormat inLogFormat`

Concurrency requirement

This function is thread safe.

Description

This function updates the `LogFormat` field of the `FPLogState` object, and determines which log format is to be used for logging.

You can initialize the default value of `LogFormat` by setting the `FP_LOGFORMAT` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPLogFormat inLogFormat`
  The log format to be used for logging — either `FP_LOGGING_LOGFORMAT_XML` or `FP_LOGGING_LOGFORMAT_TAB` (default).

Example

```
FPLogState_SetLogFormat(vLogState,
FP_LOGGING_LOGFORMAT_TAB);
```

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)

FPLogState_SetLogLevel

Syntax

```
FPLogState_SetLogLevel(FPLogStateRef ioLogState,
FPLogLevel inLogLevel)
```

Return value

`void`

Input parameters

`FPLogStateRef ioLogState, FPLogLevel inLogLevel`

Concurrency requirement

This function is thread safe.

Description

This function updates the `LogLevel` field of the `FPLogState` object. This field controls the log level used when the logger writes output. You can initialize the default value of `LogLevel` by setting the `FP_LOGLEVEL` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPLogLevel inLogLevel`
  The level at which to log the message. The following log levels are available:
  - `FP_LOGLEVEL_ERROR`: Log error messages
  - `FP_LOGLEVEL_WARN`: Log warnings and error messages
  - `FP_LOGLEVEL_LOG`: Log API calls, warnings, and errors
  - `FP_LOGLEVEL_DEBUG`: Debug log level

Example

```
FPLogState_SetLogLevel(vLogState, FP_LOGLEVEL_DEBUG);
```

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)

FPLogState_SetLogPath

Syntax

```
FPLogState_SetLogPath(FPLogStateRef ioLogState, char *inPathName)
```

Return value

```
void
```

Input parameters

```
FPLogStateRef ioLogState, char *inPathName
```

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 22.

Description

This function updates the `LogPath` field of the `FPLogState` object. This field directs the logging mechanism to write to the file specified in `inPathName`.

You can initialize the default value of the `LogPath` field by setting the `FPLogPath` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `char *inPathName`
  The string containing a full log file path.

Example

```
FPLogState_SetLogPath(vLogState, "C:\\MyLogFile.txt");
```

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_OUT_OF_MEMORY_ERR` (client error)

FPLogState_SetMaxLogSize

Syntax

```
FPLogState_SetMaxLogSize(FPLogStateRef ioLogState,
FPLong inSizeInKilobytes)
```

Return value

`void`

Input parameters

`FPLogStateRef ioLogState, FPLong inSizeInKilobytes`

Concurrency requirement

This function is thread safe.

Description

This functionupdates the `MaxLogSize` field of the `FPLogState` object, and determines the maximum size (in KB) that the log file may grow before rolling over to an overflow file. The default is 1048576 (1 GB).

You can initialize the default value of the `LogPath` field by setting the `FPLogPath` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPLong inSizeInKilobytes`
  The maximum size (in KB) that a log file may grow. For no limit on the size, specify `FP_LOG_SIZE_UNBOUNDED`.

Example

```
FPLogState_SetMaxLogSize(vLogState, 1048576); // 1 GB
```

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)

FPLogState_SetMaxOverflows

Syntax

```
FPLogState_SetMaxOverflows(FPLogStateRef ioLogState,
FPInt inCount)
```

Return value

`void`

Input parameters

`FPLogStateRef ioLogState, FPInt inCount`

Concurrency requirement

This function is thread safe.

Description

This function updates the `MaxOverflows` field of the `FPLogState` object, and determines the maximum number of backup log files to be generated (with an .# extension) if the log file size exceeds the `MaxLogSize` value. The default value is 1.

You can initialize the default value of the `MaxOverflows` field by setting the `FP_LOG_MAX_OVERFLOWS` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPInt inCount`
  The maximum number of rollover files to be generated.

Example

```
FPLogState_SetMaxOverflows(vLogState, 2);
// Save most recent 3 GB
```

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)

FPLogState_SetPollInterval

Syntax

```
FPLogState_SetPollInterval(FPLogStateRef ioLogState,
FPInt inPollIntervalInMin)
```

Return value

```
void
```

Input parameters

```
FPLogStateRef ioLogState, FPInt inPollIntervalInMin
```

Concurrency requirement

This function is thread safe.

Description

This function updates the `log state polling interval` field of the `FPLogState` object, and defines the number of minutes between config file poll events for FPLogState if a log state config file was opened. If a state file exists, it is reread and the log configuration modified accordingly at each interval.

You can initialize the default value of the `log state polling interval` field by setting the `FP_LOG_STATE_POLL_INTERVAL` environment variable.

Parameters

- `FPLogStateRef ioLogState`
  A reference to an `FPLogState` object.
- `FPInt inPollIntervalInMin`
  The number of minutes between readings of the configuration file. The default is 5 and must be greater than or equal to -1.

Example

```
FPLogState_SetPollInterval(vLogState, 5);
// Poll state file every 5 min
```

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)

Environment variables

If set, the logging environment variables listed in Table 22, "Logging environment variables,"determine the initial configuration of the `FPLogState` control class object. They also allow legacy applications to enable logging services without making code changes.

Table 22 Logging environment variables (1 of 2)

Environment variable	Description	Possible values or data type
FP_LOG_DISABLE_CALLBACK	Disables any registered log callback when set. The default is FALSE.	• TRUE or 1 • FALSE or 0 The value is not case-sensitive.
FP_LOG_MAX_OVERFLOWS	Determines the maximum number of backup (overflow) files to retain. The default is 1.	FPInt
FP_LOG_MAX_SIZE	Determines the maximum size a log file is allowed to grow (in KB). The default is `FP_LOG_SIZE_UNBOUNDED`.	FPLong
FP_LOG_STATE_PATH	Defines a path to the log state configuration file. This file contains log configuration information for controlling logging behavior. This file is polled at the interval set in `FP_LOG_STATE_POLL_INTERVAL`.	Path and file name of the log
FP_LOG_STATE_POLL_INTERVAL	Defines the interval at which a log state configuration file is polled (in minutes). If the log interval is set to 0, the SDK polls the log configuration file prior to every log event. The default is 5 (minutes).	FPInt
FP_LOGFILTER	Determines which SDK components are logged. If `FP_LOGFILTER` is set, you designate the individual components in a comma-separated list, otherwise, the default of `ALL` is used to log all components. See possible values in the adjacent column.	POOL, RETRY, XML, API, NET, TRANS, PACKET, EXCEPT, REFS, MOPI, STREAM, CSOD, CSO, MD5, APP (for application level logging), SHA, LIB (for library loading), and ALL (for all components).

Table 22 Logging environment variables (2 of 2)

Environment variable	Description	Possible values or data type

FP_LOGFORMAT	Determines the log file format—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=ProcessID.ThreadID sec=timestamp comp=component API=high level API><![CDATA[message]]></log level>`
	Tab-delimited log format per line
	where:
	`\t` = tab
	`\s` = space
	`\n` = newline
	`MS TimeStamp\tDate\sTime\t[LogLevel]\tProcessID.ThreadID\t[Component]\tMessage\n`
	`\s\sPacketInfo\n` //when applicable
		• XML
		• TAB

FP_LOGKEEP	Determines how the file is generated. 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.	• OVERWRITE • APPEND • CREATE
FP_LOGLEVEL	Determines which level of log data displays in the file. **Note:** If you do not use the FP_LOGLEVEL environment variable, the default message level is set to 3.	• `1` (Error messages only) • `2` (Warning and Error messages) • `3` (Log, Warning, and Error messages) • `4` (Debug, Log, Warning, and Error messages)
FP_LOGPATH	Defines the full path to the log file on disk. This is required to enable logging. If unset to indicate logging is disabled, this field is set to the `<NULL>` string.	Path and file name of the log Example C:\MyLog.txt

Example: XML-formatted log

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<SDKLog SDKVersion="3.1.544">

<log procID="3368" tID="1996" msec="1177102381059" usec="qu" comp="API"><![CDATA[
Start FPPool_GetComponentVersion(1,-,128)
]]></log>

<log procID="3368" tID="1996" msec="1177102381059" usec="qu" comp="API"
API="FPPool_GetComponentVersion(1,-,128)"><![CDATA[
End FPPool_GetComponentVersion(1,@version.full@,15)
]]></log>

<log procID="3368" tID="1996" msec="1177102382912" usec="qu" comp="API"><![CDATA[
Start FPPool_Open(sdk2)
]]></log>

<debug procID="3368" tID="1996" msec="1177102382912" usec="qu" comp="LIB"
API="FPPool_Open(sdk2)"><![CDATA[
Attempting to load MD5 lib: "FPMD5_Lib.dll"
]]></debug>

<warn procID="3368" tID="1996" msec="1177102382912" usec="qu" comp="LIB"
API="FPPool_Open(sdk2)"><![CDATA[
MD5 lib not found! Using internal MD5 algorithm instead.
]]></warn>

<debug procID="3368" tID="1996" msec="1177102382922" usec="qu" comp="LIB"
API="FPPool_Open(sdk2)"><![CDATA[
Attempting to load SHA256 lib: "FPSHA256_Lib.dll"
]]></debug>
.
.
.
</SDKLog>
```

Example: Tab-formatted log

Note: Because of space limitations on the page, the following example displays the 6-column log information as two 3-column segments on the page.

```
Time                  FormattedTime                LogLevel
1177082804380          2007-04-20 15:26:44.380      [none]
1177082804380          2007-04-20 15:26:44.380      [log]
1177082804380          2007-04-20 15:26:44.380      [log]
1177082806122          2007-04-20 15:26:46.122      [log]
1177082806122          2007-04-20 15:26:46.122      [debug]
1177082806142          2007-04-20 15:26:46.142      [warn]
1177082806142          2007-04-20 15:26:46.142      [debug]
1177082806142          2007-04-20 15:26:46.142      [warn]
1177082806142          2007-04-20 15:26:46.142      [debug]
1177082806142          2007-04-20 15:26:46.142      [debug]
1177082806233          2007-04-20 15:26:46.233      [log]
1177082806233          2007-04-20 15:26:46.233      [debug]

PID.Thread  Module   Message
3360.1812    [---]    SDK Version 3.1.544
3360.1812    [API]    Start FPPool_GetComponentVersion(1,-,128)
3360.1812    [API]    End FPPool_GetComponentVersion(1,@version.full@,15)
3360.1812    [API]    Start FPPool_Open(sdk7)
3360.1812    [LIB]    Attempting to load MD5 lib: "FPMD5_Lib.dll"
3360.1812    [LIB]     MD5 lib not found! Using internal MD5 algorithm instead.
3360.1812    [LIB]    Attempting to load SHA256 lib: "FPSHA256_Lib.dll"
3360.1812    [LIB]   SHA256 lib not found! Using internal SHA256 algorithm instead.
3360.1812    [LIB]    Attempting to load FPIntercept lib: "FPIntercept_g.dll"
3360.1812    [LIB]     FPIntercept lib not found! (Using all default API calls.)
3360.1812    [CORE]   Start ClusterCloud::createClusterInterfaceHelper
                     (-,-,false,true,false,-,120000,false)
3360.1812    [RETRY]  AN(sdk7) Access Role (enabled)
.
.
.
```

Example: FPLogState configuration file

```
##################################
# FPLogState Configuration File
##################################

FP_LOGPATH=C:\Logs\MyLogFile.log
FP_LOGLEVEL=4
FP_LOGFORMAT=TAB
FP_LOGFILTER=ALL
FP_LOGKEEP=OVERWRITE
FP_LOG_DISABLE_CALLBACK=TRUE
FP_LOG_STATE_POLL_INTERVAL=5
FP_LOG_MAX_SIZE=FP_LOG_SIZE_UNBOUNDED
FP_LOG_MAX_OVERFLOWS=1
```

The following format conventions apply:

- The # character defines comment lines.
- Whitespace is ignored.
- Lines are expected to be name-value pairs (delimited by =) with no spaces.
- The supported logging name-value pairs correspond directly to the defined environment variable name-values.
- The `<NULL>` string may optionally be used to indicate an "unset" value (for example, `FP_LOGPATH=<NULL>` to disable logging, or just `FP_LOGPATH=` to clear it).
- Special defines that the API supports (for example, `FP_LOG_SIZE_UNBOUNDED`) are supported as values where applicable.

12

Error Codes

This chapter describes the set of SDK error codes.

The main section in this chapter is:

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 23 on page 320, refer to a list of platform-specific error codes (for example, Windows error code 10055 means that no buffer space is available).

Note that all EMC Centera error codes are negative values.

Refer to the *EMC Centera Programmer's Guide* for more information on error handling. Also refer to "FPPool_GetLastError" on page 49 and "FPPool_GetLastErrorInfo" on page 50.

If you want to access the errors described in this section, you must include `FPErrors.h`.

Table 23 Error codes (1 of 8)

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 23 Error codes (2 of 8)

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 23 Error codes (3 of 8)

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 23 Error codes (4 of 8)

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 23 Error codes (5 of 8)

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 23 Error codes (6 of 8)

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. Allow the SDK to perform the automatic number of configured retries.
-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.
-10044	FP_STREAM_VALIDATION_ERR	A generic stream validation error occurred.
-10045	FP_STREAM_BYTECOUNT_MISMATCH_ERR	A generic stream byte count mismatch was detected.
-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.
-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 23 Error codes (7 of 8)

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.
-10164	FP_METADATA_MISMATCH_ERR	Mutable metadata mismatch found.
-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.

Table 23 Error codes (8 of 8)

Value	Error name	Description and action
-10214	FP_INVALID_XML_ERR	The specified string is not valid XML.
-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.
-10216	FP_LOGGING_CALLBACK_ERR	An error occurred in the application-defined FPLogging callback.

A

Monitoring Information

This appendix lists the syntax and provides samples of XML files that are retrieved by the MoPI functions.

The sections in this appendix are:

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

EMC 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 24 on page 343 for a complete listing of all EMC Centera alerts with their symptom codes, error levels, and detailed descriptions.

EMC 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. To meet the U.S. Department of Defense (DOD) security standards, the XML messages are encrypted and uuencoded (Unix-to-Unix encoding) using FIPs (Fair Information Practices)140 compliant encryption standards with AES (Advanced Encryption Standard) 256-bit strong encryption.

  Messages are parsed and inserted in the EMC call tracking infrastructure upon receipt. The *EMC Centera Online Help* provides more information on ConnectEMC-related CLI commands.

- ConnectEMC Notification — The system administrator can receive email notification with an HTML formatted copy of the alert message. Refer to the *EMC Centera Online Help* for more information.

- SNMP — The Simple Network Management Protocol (SNMP) is an Internet-standard protocol for managing devices on IP networks. The SNMP agent runs on all nodes with the access role and proactively sends messages called SNMP traps to a network management station. The *EMC Centera Online Help* provides more information on the SNMP-related CLI commands and SNMP details.

- EMC ControlCenter® (ECC) — As of ECC v5.2, and CentraStar v2.2, EMC ControlCenter users can monitor one or more EMC Centera clusters in their storage environment. With the introduction of ECC 5.2 SP3, the StorageScope module in ECC reports EMC Centera capacity information for clusters being monitored by ECC.

The EMC ControlCenter Storage Agent for Centera is available free of charge and is qualified to run on the EMC ControlCenter server.

- The Monitoring API (MoPI) — The SDK has the ability to receive alerts with the MoPI API call `FPEventCallback_RegisterForAllEvents` and reeive health reports via the MoPI `FPMonitor_GetDiscovery` call. "Monitoring functions" on page 266 provides more information on MoPI.

- CLI — The system administrator can download the health report, which is sent via ConnectEMC in native XML format using the show report health <*filename*> command. The CLI can be used to provide real-time reporting or can be scripted to provide scheduled reporting.

- ChargeBack Reporter — A separate application using the Centera Seek engine to provide customer capacity utilization reports based on metadata contained in the CDF.

Each alert in Table 24, "EMC Centera sensor-based alerts,"corresponds to one of the following severity-level message types:

- OK — The value of the sensor is within the normal range of operations. It is used for improvement messages only.
- Notification — Informational only.
- Warning — The value of the sensor indicates that something might be wrong or could go wrong in the near future.
- Error — Something is wrong and intervention is needed to correct the situation.
- Critical — Something is seriously wrong; urgent action is required to correct the situation.

Table 24 EMC Centera sensor-based alerts (1 of 5)

Symptom code	Severity level	Description
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 number of C-Clips that are waiting to be replicated. 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.
1.1.1.1.03.01 1.1.1.1.03.02 1.1.1.1.03.03	Warning >= 24 h Error >= 48 h Critical >= 120h	*ReplicationETA* The time it takes to replicate all parked C-Clips. The replication process on the cluster will probably take longer than 24/48/120 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.
1.1.1.1.04.01 1.1.1.1.04.02 1.1.1.1.04.03	Warning >= 24 h Error >= 48 h Critical >= 120h	*ReplicationLag* The replication process on the cluster will likely take longer than 24/48/120 hours to process the entire replication queue.
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 number of C-Clips that are waiting to be restored. 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.

Table 24 EMC Centera sensor-based alerts (2 of 5)

Symptom code	Severity level	Description
1.1.1.1.20.01 1.1.1.1.20.02 1.1.1.1.20.03 1.1.1.1.20.04 1.1.1.1.20.05 1.1.1.1.20.06 1.1.1.1.20.07 1.1.1.1.20.08	Error = paused_clusterfull Error = paused_parking_overflow Error = paused_no_capability Error = paused_authentication_failure Error = paused_pool_not_found Warning = paused_user Warning = cancelled Warning = paused_replica_feature	*ReplicationSubstate* A problem has occurred that has caused EMC Centera to automatically pause replication, or the user has paused replication.
1.1.1.1.23.01	FailState	*ReplicationQueue/Error* The replication task on a node has terminated unexpectedly.
1.1.1.1.24.01 1.1.1.1.24.02 1.1.1.1.24.03	Warning >=24 h Error >= 48 h Critical >=120 =h	*ReplicationSubstate*
1.1.2.1.02.01	Critical < 0 (1=Basic, -2=CE (Governance), -3=CE+)	*ComplianceModeRevert* Checks if the cluster has been reconfigured to a less strict compliancy more than before
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.
1.1.4.1.06.01 1.1.4.1.06.02 1.1.4.1.06.03 1.1.4.1.06.04 1.1.4.1.06.05 1.1.4.1.06.06 1.1.4.1.06.07 1.1.4.1.06.08 1.1.4.1.06.09 1.1.4.1.06.10 1.1.4.1.06.11	Error = paused_clusterfull Error = paused_parking_overflow Error = paused_no_capability Error = paused_authentication_failure Error = paused_pool_not_found Warning = paused_user Warning = cancelled Warning = paused_replica_feature Warning = paused_failed_parking Warning = paused_checkpoint_failed Warning = paused_parking_error	*RestoreState* A problem has occurred that has caused EMC Centera to automatically pause the restore, or the user has paused the restore.
1.1.11.1.01.01	Error	*DB/Disk Error Condition Exists* A non-fatal error occurred. The cause should be investigated at the earliest convenience.

Table 24 EMC Centera sensor-based alerts (3 of 5)

Symptom code	Severity level	Description
1.1.11.1.01.02	Critical	*DB/Disk Critical Condition Exists* A failure condition exists that must be investigated immediately. The cluster's integrity may be in jeopardy.
2.1.2.1.01.01	Error = INCOMPLETE	*ClusterClipIntegrity* Checks the integrity of the cluster data. 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.
2.1.2.1.03.01	Error > 0	*FailedRegenerations* At least one disk or node regeneration cannot complete or has failed.
2.1.3.1.01.01	Warning <= 0	*PowerRedundancyStatus* Power is no longer redundant on one or more nodes.
2.4.2.1.02.01	Severity = 2 Priority = 1	*DBReadOnlyCount* At least one database in the cluster is read-only.
3.1.3.1.01.01 3.1.3.1.01.02	Warning = 1 Error = 2	*InternalNICFailureCount* The number of failed internal NICs. 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.

Table 24 EMC Centera sensor-based alerts (4 of 5)

Symptom code	Severity level	Description
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 > 1	*RootSwitchFailures* The number of failed root switches. 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.
3.1.4.1.03.01	Error > 0	*ExternalSwitchFailure* The majority (more than 50% and at least three) of the nodes with the access role report simultaneous external network interface card (NIC) failures.
3.1.5.1.01.01	Warning = 1	*DiskOffline* The number of offline disks. 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 = 1	*NodesOffline* The number of offline nodes. 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.

Table 24 EMC Centera sensor-based alerts (5 of 5)

Symptom code	Severity level	Description
4.1.1.1.01.01 4.1.1.1.01.02	Error > 85 °C	*CPUTemperature* CentraStar has determined that 1 or more nodes in the cluster has a running temperature higher than 85 °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.
4.1.1.1.02.01	Error > 70%	*VarPartitionUsedSpace* The amount of used space on the partition. CentraStar has determined that 1 or more nodes in the cluster with an internal partition used by CentraStar has 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%	*AvailableCapacityPercent* If the regeneration buffer > 0, then it shows 999. If the regeneration buffer < 0, then it shows the Safe Free Capacity divided by the Rack Total Capacity.
5.2.2.1.04.01 5.2.2.1.04.02 5.2.2.1.04.03	Warning < 20% Error <10% Critical <= 0%	*FreeObjectPercent* The percentage of Free Object Count for the cluster.
5.2.5.7.02.01	Disabled	*FeedOrganicSingleStep* An internal primary fragment migration task on one or more nodes has run out of space and stopped its processing.
5.2.7.1.02.01	Error, if not empty	*FailingNodeUpgrades* At least one node has failed an upgrade process.

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
$Hardware.Main.NIC.Main.InternalNICFailureCount@$CLUSTER
:eth0
Component: node
Component ID: c001n05
Sub-Component: nic
Sub-Component ID: eth0
```

B

Deprecated Functions

This appendix lists deprecated EMC 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 25 on page 350 lists deprecated EMC Centera API functions.

Table 25 Deprecated functions

Function name	Deprecated release	Comments
FPMonitor_GetDiscovery	3.2	Use the FPMonitor_GetDiscoveryStream function (page 276).
FPMonitor_GetAllStatistics	3.2	Use the FPMonitor_GetAllStatisticsStream function (page 273).
FPTime_LongToString	3.1	Use the FPTime_SecondsToString or FPTime_MillisecondsToString function. Refer to "Time functions" on page 280.
FPTime_StringToLong	3.1	Use the FPTime_StringToSeconds or FPTime_StringToMilliseconds function. Refer to "Time functions" on page 280.
FPClip_Purge	2.3	Use the FP_OPTION_DELETE_PRIVILEGED option of FPClip_AuditedDelete() (page 73) 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 238.
FPQuery_GetPoolRef	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to "Query functions" on page 238.
FPQuery_FetchResult	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to "Query functions" on page 238.
FPQuery_Close	2.3	Use the FPQueryExpression, FPPoolQuery, and FPQueryResult functions. Refer to "Query functions" on page 238.
FPStream_CreateWithBuffer	1.1	Use FPStream_CreateBufferForInput( ) (page 203) and FPStream_CreateBufferForOutput( ) (page 204).
FPStream_CreateWithFile	1.1	Use FPStream_CreateFileForInput( ) (page 205) and FPStream_CreateFileForOutput( ) (page 206).
FPStream_Read	1.1	Use the generic stream facility. Refer to "Stream functions" on page 200.
FPStream_Write	1.1	Use the generic stream facility. Refer to "Stream functions" on page 200.
FPStream_SetMarker	1.1	Use the generic stream facility. Refer to "Stream functions" on page 200.
FPStream_GetMarker	1.1	Use the generic stream facility. Refer to "Stream functions" on page 200.

Deprecated options

Table 26 on page 351 lists deprecated EMC Centera API options.

Table 26 Deprecated options

Option name	Deprecated release	Comments
FP_OPTION_ENABLE_DUPLICATE_DETECTION	2.1	Was used by FPTag_BlobWrite( ).
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 Programming 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 EMC 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 EMC 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 EMC 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 EMC 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 EMC 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 EMC 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 EMC 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

G

I

J

L

M

N

O

P

X